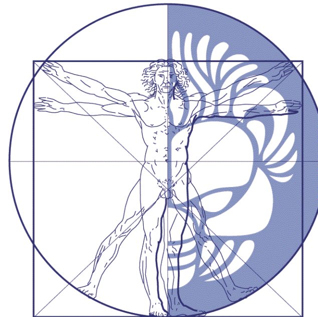
Università degli Studi Mediterranea di Reggio Calabria

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Elettronica

Dipartimento di Informatica, Matematica, Elettronica e Trasporti



Tesi di Laurea

Progettazione e implementazione di attività di "Code Inspection" su un software per la gestione degli utenti di un'azienda di telecomunicazioni

Relatori

Prof. Domenico Ursino

Ing. Francesco Profazio

Ing. Salvatore Pullano

Candidato

Santo Strati

Anno Accademico 2007-2008

Eppure, non volevo tentar di vivere se non ciò
che spontaneamente voleva erompere da me.
Perché era tanto mai difficile?

Hermann Hesse

Indice

Elenco delle figure VIII

Elenco delle tabelle XI

Introduzione 1

Parte I Background culturale

1 La Sicurezza Informatica e la Sicurezza nel Codice 9
1.1 Introduzione alla Sicurezza Informatica 9
1.2 I concetti di “Qualità” e “Sicurezza” del Software 10
1.2.1 Considerazioni introduttive 10
1.2.2 Software: “Qualità” vs. “Sicurezza” 11
1.2.3 Controllo di problemi di qualità e sicurezza 12
1.3 Principali vulnerabilità dei Sistemi Informativi dovuti a errori di programmazione 13
1.3.1 Validazione dell’Input 13
1.3.2 Bound Checking e problematiche di overflow 19
1.3.3 Session Management 23
1.3.4 Crittografia 25
1.3.5 Error e Time Handling 28
1.3.6 Processi di Tracciamento 32
1.4 Principi di Sicurezza per il Codice Sorgente 34
1.4.1 Origine delle Vulnerabilità 34
1.4.2 Importanza di una Programmazione Sicura 36
1.4.3 Progettazione e Sviluppo in Sicurezza dell’Applicazione 36

2 La Code Inspection 45
2.1 Approccio al Problema della Sicurezza del Software 45
2.2 Scopo del Processo di Code Inspection 46
2.3 Descrizione del Processo di Code Inspection 47
2.3.1 Planning 48
2.3.2 Overview 48
2.3.3 Preparation 48
2.3.4 Inspection Meeting 49
2.3.5 Rework 49
2.3.6 Follow-up 49
2.4 Strumenti Software 50
2.4.1 Coverity Prevent 50
2.4.2 Fortify Source Code Analysis Suite 51
2.4.3 Secure Software CodeAssure 52
2.4.4 Klocwork K7 52
2.4.5 Ounce Labs Prexis 53
2.5 Requisiti e vincoli 54
2.5.1 Requisiti tecnologici 54
2.5.2 Requisiti organizzativi 54
2.6 La certificazione del codice eseguibile 55
2.6.1 Compilazione in ambiente interno/controllato 56
2.6.2 Ispezione del decompilato 56
2.6.3 Ispezione a campione 57
2.6.4 Cenni sulla ricerca delle invarianti 57
2.6.5 Comparazione a campione negli ambienti di origine 58
2.6.6 Considerazioni conclusive 58

3 Il software Fortify 59
3.1 Introduzione 59
3.2 Fortify Source Code Analysis Suite 60
3.2.1 Fortify Source Code Analyzer 62
3.2.2 Fortify Audit Workbench 63
3.2.3 Fortify Manager 66
3.2.4 Secure Coding Rulepacks 68
3.2.5 Rules Builder 69

Parte II Progettazione delle attività di code inspection

4 Descrizione della realtà di riferimento 75
4.1 Premessa 75
4.2 Ambiente di riferimento 76
4.2.1 Cenni sul processo di sviluppo delle applicazioni 77
4.2.2 Volumi in gioco 77
4.2.3 Caratteristiche particolari 78
4.3 Strutturazione del processo di code inspection nell'ambito di riferimento 79
4.3.1 Verifica formale della completezza del materiale e della documentazione 82
4.3.2 Caricamento del materiale sulla piattaforma di analisi 86
4.3.3 Individuazione delle aree di esposizione dell'applicazione . . . 86
4.3.4 Selezione delle firme di rilevamento 87
4.3.5 Identificazione delle occorrenze e classificazione delle vulnerabilità tecnologiche rilevate 88
4.3.6 Contestualizzazione delle vulnerabilità 89
4.3.7 Organizzazione dei risultati 91

5 Analisi degli interventi di code inspection richiesti 95
5.1 Premessa 95
5.2 Interventi di code inspection richiesti 96
5.3 Traduzione di codice C/C++ 96
5.3.1 Integrazione con Make 97
5.4 Traduzione di codice Java 97
5.4.1 Traduzione di applicazioni J2EE 98
5.5 Attività preliminari 98
5.5.1 Firme di rilevamento Fortify 99
5.5.2 Associazione tra le aree di esposizione dell'applicazione e le famiglie di firme Fortify 101
5.5.3 Metodologia per la selezione delle firme di rilevamento 102

6 Sviluppo in sicurezza del codice C/C++ 105
6.1 Introduzione ai linguaggi C/C++ 105
6.2 Best practices considerate per l'attività di code inspection 106
6.2.1 Dichiarazioni 106
6.2.2 Inizializzazioni 107
6.2.3 Utilizzo dei tipi di dati 107
6.2.4 Macro 109
6.2.5 L'operatore `sizeof` ed il passaggio di dati come parametri . 110
6.2.6 Allocazione dinamica 110
6.2.7 Deallocazione 111
6.2.8 Puntatori 112
6.2.9 Casting e problematiche di gestione delle variabili numeriche 112

6.2.10 Condizioni . . . 113
6.2.11 Controllo del flusso . . . 114
6.2.12 Passaggio di argomenti . . . 115
6.2.13 Valori di ritorno . . . 115
6.2.14 Chiamate a funzioni . . . 115
6.2.15 File . . . 115
6.2.16 Gestione degli errori . . . 115
6.2.17 Sicurezza dell'applicazione . . . 116
6.3 Vulnerabilità per i linguaggi di programmazione C/C++ riscontrabili tramite Fortify . . . 116

7 Sviluppo in sicurezza del codice Java . . . 125
7.1 Introduzione al linguaggio Java . . . 125
7.2 Best practices considerate per l'attività di code inspection . . . 127
7.2.1 Inizializzazioni . . . 127
7.2.2 Visibilità . . . 129
7.2.3 Modificatori . . . 130
7.2.4 Utilizzo degli oggetti mutevoli . . . 130
7.2.5 Definizione delle classi . . . 131
7.2.6 Firma del codice e permessi speciali . . . 132
7.2.7 Utilizzo dei comandi di sistema . . . 132
7.2.8 Oggetti . . . 132
7.2.9 Serializzazione e deserializzazione . . . 133
7.2.10 Memorizzazione delle informazioni riservate . . . 134
7.2.11 Package . . . 135
7.2.12 Gestione delle eccezioni . . . 135
7.3 Applet Java . . . 137
7.3.1 Informazioni critiche . . . 138
7.3.2 Visibilità . . . 138
7.3.3 Overriding . . . 138
7.3.4 Modificatori . . . 138
7.3.5 Firma delle Applet . . . 138
7.3.6 Validazione delle informazioni trasferite . . . 138
7.3.7 Utilizzo delle Applet di terze parti . . . 139
7.4 Servlet Java . . . 139
7.4.1 Utilizzo delle richieste di tipo HTTP POST e HTTP GET . 139
7.4.2 Filtraggio dell'input . . . 139
7.4.3 Utilizzo dei Web Application Firewall . . . 141
7.4.4 Proteggere i dati personali e/o sensibili . . . 141
7.4.5 Switching tra le modalità SSL e non-SSL . . . 141
7.4.6 Statement SQL . . . 141
7.4.7 Token di sessione . . . 142
7.4.8 Cookie . . . 142

7.4.9 Limitare la dimensione delle risposte HTTP 143
7.4.10 HTTP Referer 143
7.4.11 Trattamento dei file e degli oggetti embedded 143
7.4.12 Corretta gestione degli errori e delle eccezioni 143
7.4.13 Limitare l'utilizzo delle risorse macchina 144
7.5 Vulnerabilità per il linguaggio di programmazione Java riscontrabili tramite Fortify 144

Parte III Implementazione e validazione delle attività di code inspection

8 Selezione delle firme di rilevamento delle vulnerabilità relative alle best practices del codice C/C++ 161
8.1 Introduzione 161
8.2 Selezione delle firme per il linguaggio C/C++ 162
8.2.1 Input Source 163
8.2.2 Control Flow 165
8.2.3 Data Flow 168
8.2.4 Internal 169
8.2.5 Semantic 169
8.2.6 Structural 171
8.3 Firme ad hoc 172

9 Selezione delle firme di rilevamento delle vulnerabilità relative alle best practices del codice Java 173
9.1 Introduzione 173
9.2 Selezione delle firme per il linguaggio Java 173
9.2.1 Input Source 175
9.2.2 Control Flow 177
9.2.3 Data Flow 178
9.2.4 Internal 180
9.2.5 Semantic 181
9.2.6 Structural 182
9.3 Firme ad hoc 183

10 Validazione della selezione delle firme di rilevamento 185
10.1 Introduzione 185
10.2 Considerazioni relative all'attività di selezione delle firme 186
10.3 Metodologia 188
10.4 Codice 1 189
10.4.1 Verifica formale della completezza del materiale e della documentazione 189

10.4.2 Caricamento del materiale sulla piattaforma di analisi 189
10.4.3 Attività di Analisi Preliminare 192
10.4.4 Attività di Analisi Avanzata 197
10.4.5 Audit con selezione delle firme relative alle "best practices" per il "Codice 1" 199
10.5 Codice 2 203
10.5.1 Audit con selezione delle firme relative alle "best practices" per il "Codice 2" 204

Parte IV Conclusioni

11 Conclusioni 215

12 Uno sguardo al futuro 217

Riferimenti bibliografici 219

Firme di rilevamento del software Fortify relative alle vulnerabilità riscontrabili nei codici sorgente C/C++ 221

Firme di rilevamento del software Fortify relative alle vulnerabilità riscontrabili nei codici sorgente Java 229

Elenco delle figure

1.1 Esempio di script vulnerabile a Shell Execution Command 15
1.2 Esempio di servlet vulnerabile al Cross Site Scripting 16
1.3 Esempio di script vulnerabile a SQL Injection 18
1.4 Rappresentazione generica di uno Stack Overflow 20
1.5 Esempio di default script Web soggetto a Information Disclosure . . 30
1.6 Esempio di Directory Listing 30

2.1 Diagramma di flusso di una Fagan Inspection 48
2.2 Diagramma di flusso di un semplice modello di analisi del codice sorgente 51

3.1 Struttura del Fortify Source Code Analysis Suite 61
3.2 Interfaccia principale del Fortify Audit Workbench 64
3.3 Risultati di un'analisi tramite Fortify Manager 68

4.1 Processo di Code Inspection 81

8.1 Selezione delle firme di rilevamento a partire dalle best practices . . 164
8.2 Rulepack Management: Schermata principale 165
8.3 Rulepack Management: Selezione firme 166
8.4 Selezione delle firme di rilevamento relative a "inputsource" 166
8.5 Selezione delle firme di rilevamento relative all'analizzatore Control Flow 167
8.6 Selezione delle firme di rilevamento relative all'analizzatore Data Flow 169
8.7 Selezione delle firme di rilevamento relative all'analizzatore Internal 170
8.8 Selezione delle firme di rilevamento relative all'analizzatore Semantic 171
8.9 Selezione delle firme di rilevamento relative all'analizzatore Structural 172

9.1 Rulepack Management: Schermata principale 175
9.2 Rulepack Management: Selezione firme 176
9.3 Selezione del pacchetto di firme “Fortify Secure Coding Rules, Extended, Java” 176
9.4 Selezione delle firme di rilevamento relative a “inputsource” 177
9.5 Selezione delle firme di rilevamento relative all’analizzatore Control Flow 178
9.6 Selezione delle firme di rilevamento relative all’analizzatore Data Flow 180
9.7 Selezione delle firme di rilevamento relative all’analizzatore Internal 181
9.8 Selezione delle firme di rilevamento relative all’analizzatore semantico 182
9.9 Selezione delle firme di rilevamento relative all’analizzatore strutturale 183

10.1 Selezione delle firme di rilevamento a partire dalle “best practices” 186
10.2 Funzione g 187
10.3 Funzione f 187
10.4 Vulnerabilità identificate nel “Codice 1” con selezione delle firme “Broad” di default per Fortify 195
10.5 Vulnerabilità identificate nel “Codice 1” con selezione delle firme “Medium” di default per Fortify 196
10.6 Vulnerabilità identificate nel “Codice 1” con selezione delle firme “Targeted” di default per Fortify 196
10.7 Visualizzazione del progetto tramite Fortify Security Manager 198
10.8 Abilitazione di tutte le firme del pacchetto “Fortify Secure Coding Rules, Extended, JSP” 199
10.9 Vulnerabilità identificate nel “Codice 1” considerate “Hot” con la selezione delle firme relativa alle “best practices” 200
10.10 Vulnerabilità identificate nel “Codice 1” considerate “Warning” con la selezione delle firme relativa alle “best practices” 201
10.11 Vulnerabilità identificate nel “Codice 1” considerate “Info” con la selezione delle firme relativa alle “best practices” 201
10.12 Selezione della firma “Java Properties” 202
10.13 Grafico comparativo delle selezioni per il “Codice 1” 203
10.14 Vulnerabilità identificate nel “Codice 2” con selezione delle firme “Broad” di default per Fortify 204
10.15 Vulnerabilità identificate nel “Codice 2” con selezione delle firme “Medium” di default per Fortify 205
10.16 Vulnerabilità identificate nel “Codice 2” con selezione delle firme “Targeted” di default per Fortify 205
10.17 Vulnerabilità identificate nel “Codice 2” considerate “Hot” con la selezione delle firme relativa alle “best practices” 206

10.18 Vulnerabilità identificate nel "Codice 2" considerate "Warning" con la selezione delle firme relativa alle "best practices" 207
10.19 Vulnerabilità identificate nel "Codice 2" considerate "Info" con la selezione delle firme relativa alle "best practices" 207
10.20 Grafico comparativo delle selezioni per il "Codice 2" 208

Elenco delle tabelle

4.1 Informazioni necessarie all'analisi 84
4.2 Albero delle vulnerabilità 87
4.3 Tassonomia delle vulnerabilità 90
4.4 Informazioni contenute nel "Rapporto di Verifica" 92
4.5 Informazioni contenute nel "Riepilogo delle Vulnerabilità Riscontrate" 92

5.1 Associazione tra aree di esposizione e famiglie Fortify 102

6.1 C/C++: Dimensionamento di un array 107
6.2 C/C++: Costanti dichiarate tramite la parola chiave `const` 107
6.3 C/C++: Stringhe e carattere `NULL` 108
6.4 C/C++: Buffer e problematiche di bound-checking 109
6.5 C/C++: Buffer e problematiche di format string overflow 110
6.6 C/C++: Macro ed effetti collaterali 110
6.7 C/C++: Operatore `sizeof()` 111
6.8 C/C++: Cancellazione di array 111
6.9 C/C++: Gestione di puntatori a `NULL` 112
6.10 C/C++: Comparazione signed/unsigned tra interi 113
6.11 C/C++: Conversione fra interi da evitare 114
6.12 C/C++: Variabili di controllo e loro limiti 115

7.1 Java: Gestione dinamica delle allocazioni/deallocazioni di memoria 129
7.2 Java: Un primo esempio dell'utilizzo di oggetti mutevoli 131
7.3 Java: Un secondo esempio dell'utilizzo di oggetti mutevoli 131
7.4 Java: Utilizzo di classi interne 131
7.5 Java: Comandi di sistema 133
7.6 Java: Comparazione degli oggetti 134
7.7 Java: Gestione delle eccezioni nel caso di input nullo 136

7.8 Java: Utilizzo della clausola throws 137
7.9 Sostituzioni dei caratteri speciali da effettuare nel codice HTML . . . 140

10.1 Vulnerabilità riscontrate con varie selezioni per il "Codice 1" 202
10.2 Vulnerabilità riscontrate con varie selezioni per il "Codice 2" 208

Introduzione

Da sempre la conoscenza, intesa come apprendimento di informazioni utili, ha portato con sé vantaggi di varia natura.

Per questo motivo, sin dall'antichità, regnanti e generali proteggevano con molteplici stratagemmi le loro comunicazioni per governare sulle proprie terre e comandare i propri eserciti. La caduta di preziose informazioni in mano nemica avrebbe provocato effetti disastrosi.

L'importanza dell'informazione non è mutata con il passare del tempo. Sono cambiati, invece, il modo in cui viene trasmessa e la capacità con cui viene elaborata (nonché i supporti sui quali viene codificata).

Fino alla scoperta del telegrafo ottico da parte di Charles Chappe, sul finire del XVIII secolo, le informazioni viaggiavano alla stessa velocità delle persone. Mentre, nella nostra epoca dell'ICT (*Information and Communication Technology*), esse viaggiano circa un milione di volte più velocemente delle persone (persone circa 1000 Km/h, informazioni circa 300.000 Km/s $= 10,8 \times 10^8$ Km/h).

Dunque, il nostro tempo è caratterizzato da una velocità istantanea delle comunicazioni e da una sempre crescente capacità di elaborazione delle informazioni; queste ultime, di conseguenza, crescono a dismisura (basti pensare ai tanto attuali contenuti in alta definizione).

I calcolatori, e loro ormai imprescindibili reti di interconnessione, divengono sempre più uno strumento indispensabile in ambiti come la ricerca scientifica, lo sviluppo, la progettazione, l'azione di compravendita finanziaria, la comunicazione personale, e ogni livello del business aziendale. Man mano che la rivoluzione delle comunicazioni trasforma la società, l'*arte* di proteggere le informazioni diventa sempre più rilevante per la vita di tutti i giorni. Lo sviluppo di sistemi di sicurezza informatica può essere considerato un forma di lotta per la sopravvivenza. Le informazioni sono esposte alle insidie di persone che, per mestiere o vocazione, fanno di tutto per impadronirsene. Quando scovano una nuova tecnica che sfrutta un punto debole del sistema, quest'ultimo diventa inutile, cadendo in disuso o

evolvendo in una nuova struttura più sicura. Quest'ultima prospera finché gli eventuali attaccanti non scoprono una nuova falla, e così via. Di conseguenza, il concetto di sistema informativo, sempre più automatizzato, è complementare a quello di sicurezza informatica.

Secondo l'istituto di ricerca Gartner, tre incidenti di sicurezza su quattro avvengono a livello applicativo. Appare evidente, quindi, che il software, oltre ad essere di qualità, debba essere sicuro. Quest'ultimo obiettivo può essere perseguito attraverso un'attenta osservanza dei principi guida per una programmazione sicura. Troppo spesso, purtroppo, tali principi sono disattesi e, quindi, al fine di ridurre il numero di vulnerabilità critiche, si adotta un processo di ispezione del codice sorgente (*Code Inspection*).

I software di analisi del codice sorgente (*Source Code Analyzers*, SCA) hanno conosciuto vasta diffusione solo negli ultimi anni. Essi sono in grado, sulla base della collezione e dell'utilizzo di specifiche informazioni in merito a svariate tipologie di vulnerabilità, di individuare le porzioni di codice critiche. In tale contesto, per perseguire le finalità di cui sopra, sono stati realizzati vari pacchetti software, quasi esclusivamente di carattere commerciale.

All'interno di tali strumenti un ruolo fondamentale è ricoperto dalle cosiddette "firme di rilevamento", un insieme di regole opportunamente studiate, definite e implementate. Dalla selezione e dall'applicazione opportune delle firme dipende la sicurezza dell'applicativo software, e non solo; infatti, la compromissione di un software potrebbe coinvolgere tutte le risorse ad esso collegate.

La presente tesi si colloca in questo particolare contesto. Sulla base di una collaborazione con l'azienda "ACSI Informatica" di Roma, è stato possibile rapportarsi direttamente con il mondo della sicurezza dei sistemi informatici relativi ad un'azienda leader nazionale nel campo delle telecomunicazioni.

Gli strumenti software di ispezione del codice sorgente hanno costituito il nostro principale strumento di lavoro. Particolare considerazione, successivamente ad una fase di studio generale sui principali prodotti presenti nel mercato, è stata data ad una delle migliori tecnologie sviluppate in questo settore, ovvero al software Fortify Source Code Analysis Suite (in breve, Fortify SCAS).

Le attività svolte nell'ambito della presente tesi, strettamente relazionate al particolare contesto di riferimento, hanno avuto principalmente lo scopo di studiare e implementare firme di rilevamento di vulnerabilità relative al software Fortify.

Dopo un'approfondita analisi del contesto di riferimento, è stata effettuata un'opportuna selezione di firme di rilevamento; successivamente, ci si è concentrati sulla validazione della suddetta selezione attraverso la realizzazione di un processo di auditing.

La selezione delle firme di rilevamento delle vulnerabilità è avvenuta in due fasi. La prima è stata volta alla selezione delle firme di rilevamento che rispecchiano le "best practices" dei linguaggi di programmazione C/C++; la seconda, invece,

è stata volta alla selezione delle firme di rilevamento che rispecchiano le "best practices" del linguaggio di programmazione Java.

È stato particolarmente interessante partecipare ad un primo stadio sperimentale di un'attività lavorativa, volta all'analisi completa di un codice sorgente richiesto da un committente che, come detto in precedenza, è una delle grandi aziende leader nazionali nel campo delle telecomunicazioni. Le attività connesse con il presente lavoro di tesi si sono concluse con la valutazione di tutte le esperienze effettuate attraverso un processo di auditing, con la collezione di informazioni circa l'attività interna di un sistema e con l'analisi delle stesse al fine di scoprire violazioni della sicurezza e di diagnosticarne le cause.

La presente tesi, strutturata in quattro parti, ha lo scopo di illustrare quanto finora espresso.

La Parte I fornirà l'insieme delle nozioni di base necessarie, in particolare, per la comprensione delle metodologie e delle tecnologie utilizzate. Essa è suddivisa in tre capitoli.

Il Capitolo 1 sarà interamente dedicato all'introduzione del lettore ai concetti che costituiscono le fondamenta della sicurezza informatica. Dopo aver distinto i concetti di "qualità" e "sicurezza" del software, verranno illustrate le principali vulnerabilità dovute ad errori di programmazione, le loro origini ed i principi di sicurezza volti ad evitarle.

Il Capitolo 2 illustrerà l'approccio al problema della sicurezza del software attraverso la descrizione del processo di ispezione del codice sorgente, sottolineandone le finalità e trattando i software più utilizzati; il capitolo si chiuderà evidenziando quali caratteristiche dovrebbero possedere tali software e discutendo alcune possibili soluzioni al problema della certificazione del codice eseguibile.

Il Capitolo 3 introdurrà il software Fortify Source Code Analysis Suite nelle sue edizioni e componenti costitutive.

Nella Parte II, suddivisa in quattro capitoli, verranno introdotte la realtà di riferimento, le analisi d'intervento richieste e le linee guida per lo sviluppo di codice in sicurezza per i linguaggi di programmazione più utilizzati, cioè C/C++ e Java.

Il contesto aziendale di riferimento per lo svolgimento delle nostre attività verrà introdotto nel Capitolo 4; in questo capitolo verrà mostrata anche la strutturazione del processo di ispezione in questa particolare realtà.

Il Capitolo 5 presenterà le attività di code inspection richieste; verranno anche introdotte alcune metodologie e nozioni propedeutiche per il prosieguo.

Il Capitolo 6 illustrerà una serie di "best practices" per lo sviluppo di codice sicuro C/C++; relativamente a questi linguaggi, il capitolo si chiuderà descrivendo le categorie di vulnerabilità riscontrabili tramite il software Fortify.

Analogamente al capitolo precedente, il Capitolo 7 illustrerà una serie di "best practices" per lo sviluppo di codice sicuro Java (comprese Applet e Servlet);

relativamente a questo linguaggio, il capitolo si chiuderà descrivendo le categorie di vulnerabilità riscontrabili tramite il software Fortify.

La selezione delle firme di rilevamento relative alle pratiche di programmazione sicura trattata nella parte precedente sarà sviluppata all'interno della Parte III. Essa è suddivisa in tre capitoli.

Il Capitolo 8 mostrerà la selezione delle firme di rilevamento delle vulnerabilità relative alle "best practices" del codice C/C++, mentre il Capitolo 9 mostrerà la selezione delle firme di rilevamento delle vulnerabilità relative alle "best practices" del codice Java.

Il Capitolo 10 illustrerà l'esperienza di analisi completa di un codice sorgente Java/JSP; successivamente questo stesso codice, insieme ad un altro, sarà oggetto del processo di auditing al fine di validare la qualità della selezione delle firme di rilevamento effettuata.

La Parte IV, che conclude la tesi, è costituita dai Capitoli 11 e 12. Il primo trarrà le opportune conclusioni mentre il secondo indicherà alcuni possibili sviluppi futuri.

Parte I

Background culturale

In questa prima parte si intende introdurre il lettore alle nozioni che rappresentano le basi per la piena comprensione della presente tesi. In particolare, attraverso il Capitolo 1, si intendono presentare i principi chiave della sicurezza informatica, evidenziando la differenza tra la qualità e la sicurezza del software; il capitolo terminerà con una panoramica sulla problematica della sicurezza nel codice sorgente. Il Capitolo 2 illustrerà l'approccio al problema della sicurezza del software attraverso la descrizione del processo di Code Inspection e degli strumenti software più utilizzati, evidenziando quali caratteristiche dovrebbero possedere questi ultimi; questo capitolo si chiuderà discutendo alcune possibili soluzioni al problema della certificazione del codice eseguibile. Il Capitolo 3 introdurrà il software Fortify Source Code Analysis Suite ed i cinque componenti che ne costituiscono l'edizione completa.

1

La Sicurezza Informatica e la Sicurezza nel Codice

Il presente capitolo ha lo scopo di introdurre il lettore al concetto di sicurezza informatica, ai principi che ne stanno alla base e, successivamente, ai principali aspetti legati al codice sorgente che caratterizza gli applicativi software. In questo ambito, si evidenzierà la differenza tra i concetti di qualità e sicurezza e si fornirà una panoramica sulle principali vulnerabilità dovute ad errori di programmazione. Nella sua parte finale, si punterà l'attenzione sugli accorgimenti necessari per garantire la sicurezza del codice sorgente.

1.1 Introduzione alla Sicurezza Informatica

Con il termine di *sicurezza informatica* si intende quell'insieme di sistemi di protezione che consentono alle organizzazioni di minimizzare la quantità e la pericolosità delle minacce rivolte alla parte automatizzata del proprio sistema informativo. La sicurezza di un sistema di informazioni, in particolare di un sistema informatico, ruota attorno a tre principi chiave che sono: *riservatezza*, *integrità* e *disponibilità* (C-I-A, Confidentiality, Integrity and Availability). A seconda dell'applicazione di interesse, o del contesto di utilizzo, uno di questi principi potrebbe essere considerato maggiormente rilevante rispetto agli altri.

La *riservatezza* è legata all'atto di prevenire una divulgazione di informazioni sensibili non autorizzata. Tale divulgazione potrebbe essere intenzionale, ad esempio dovuta alla risoluzione di un codice di cifratura e, di conseguenza, alla possibilità di lettura delle informazioni, o non intenzionale, dovuta alla negligenza o all'incompetenza nella gestione delle informazioni da parte di individui interessati.

Per quanto concerne l'*integrità*, sono da individuarsi tre suoi principali fini:

- Prevenzione rispetto all'apporto di modifiche alle informazioni da parte di utenti non autorizzati.

- Prevenzione rispetto all'apporto non autorizzato o non intenzionale di modifiche alle informazioni da parte di utenti autorizzati.
- Conservazione della consistenza, interna ed esterna; più specificatamente:
  - La consistenza interna assicura che i dati interni siano consistenti. Per esempio, in un database di un'organizzazione, il numero totale di articoli posseduti deve essere uguale alla somma degli stessi articoli mostrata nel database.
  - la consistenza esterna assicura che i dati memorizzati in un database siano consistenti con il mondo reale. Ritornando all'esempio precedente, il numero totale di articoli fisicamente presenti su un ipotetico scaffale deve equivalere al numero di articoli registrati dal database per quello stesso scaffale.

La *disponibilità* assicura che un utilizzatore autorizzato del sistema abbia accesso alle informazioni nello stesso sistema e verso la rete in modo continuo e opportuno dal punto di vista temporale.

Ulteriore considerazione meritano, rispetto al concetto di sicurezza informatica, i seguenti quattro termini, certamente relazionati a quanto scritto finora nei C-I-A:

- *Identificazione*: atto attraverso cui un utente dichiara la propria identità rispetto ad un sistema.
- *Autenticazione*: verifica della validità dell'identità dichiarata da un utente, ad esempio, attraverso l'utilizzo di una password.
- *Accountability*: determinazione delle azioni e del comportamento di un singolo individuo all'interno di un sistema e conservazione delle responsabilità individuali.
- *Autorizzazione*: insieme dei privilegi assegnati ad un individuo (o ad un processo) che permettono l'accesso a determinate risorse del sistema.

1.2 I concetti di "Qualità" e "Sicurezza" del Software

1.2.1 Considerazioni introduttive

Comunemente si pensa che i concetti di qualità del software e di sicurezza del software siano simili, o addirittura uguali. Tuttavia, ad una più attenta analisi, osserviamo che i due concetti sono molto differenti; più specificatamente, si può affermare che la qualità del software è cumulativa, mentre la sicurezza del software è assoluta. La qualità del software è cumulativa in quanto, ai fini del rilascio di un sistema software, è possibile accettare un certo numero di difetti. La sicurezza del software, invece, è assoluta in quanto una singola vulnerabilità nell'applicazione potrebbe, in ultima istanza, provocare effetti catastrofici.

Gli strumenti che identificano i problemi di qualità devono essere estremamente accurati e, fino a un certo punto, completi. Gli strumenti che identificano i problemi di sicurezza, invece, devono poter individuare ogni possibile vulnerabilità con un'accuratezza adeguata. In aggiunta, gli strumenti usati per rendere il software sicuro dovrebbero essere adattabili agli esistenti cicli di sviluppo/rilascio di un'organizzazione.

1.2.2 Software: “Qualità” vs. “Sicurezza”

La qualità del software ha assunto negli anni un interesse sempre crescente; essa ha rappresentato una parte integrante dello sviluppo del software sin dai primi utilizzi di linguaggi ad alto livello. Più recentemente, dal momento in cui gli applicativi software sono stati aperti al “mondo esterno” attraverso Internet, extranet e applicazioni di commercio elettronico, le organizzazioni che gestivano questi sistemi hanno posto sempre maggiore attenzione alla sicurezza del software. Chi ha poca dimestichezza con la sicurezza del software può credere che la qualità e la sicurezza del software siano la stessa cosa. Non è così.

Un problema di qualità del software può essere definito come *il fallimento, da parte di un pezzo di codice, nel realizzare delle specifiche richieste.* Ciò può portare, da un lato, a credere che l'applicazione potrebbe funzionare differentemente, dall'altro, che il codice potrebbe fallire nel rispondere a una connessione di rete basata su un ben determinato protocollo.

Una vulnerabilità nella sicurezza del software, invece, può essere definita come una *caratteristica del codice che permette di compromettere il suo funzionamento specifico da parte di un attaccante.*

A prima vista, sembrerebbe che un attaccante si limiti a sfruttare i problemi di qualità del software. In realtà, la maggior parte delle vulnerabilità nella sicurezza risiedono nel codice e sono assolutamente accettabili in un'ottica di qualità.

I problemi di qualità e sicurezza possono, occasionalmente, intersecarsi. Per esempio, ci sono volte in cui i problemi di qualità nel codice sorgente dell'applicazione potrebbero portare a problemi di sicurezza se opportunamente sollecitati.

Qualità e sicurezza sono simili nel fatto di essere entrambe caratteristiche o attributi olistici. Per ogni sistema complesso è impossibile dichiarare la sua qualità esaminandone un solo lato. Solo quando si valuta il sistema nella sua interezza si può determinare la sua qualità. Prendiamo, per esempio, una nuova casa costruita con i migliori materiali disponibili. Ad un controllo rigoroso, un eventuale acquirente potrebbe ritenere carente l'illuminazione nel bagno degli ospiti e notare che la mansarda ha le tegole di vinile piuttosto che di ceramica. Queste caratteristiche, pur non essendo perfette, non riducono la valutazione complessiva della qualità da parte dell'acquirente. Questo illustra la natura cumulativa della qualità.

Un simile scenario si ha con il software. Una moderna applicazione di large business intelligence può contenere dozzine o centinaia di difetti, ma pochi di que-

sti avranno un impatto significativo sulle prestazioni dell'applicazione avvertibili dagli utenti. Tutti coloro che utilizzano l'applicazione reputeranno che il relativo codice sia di qualità. Quindi, relativamente al software, la qualità è la somma totale di tutte le componenti.

Come con la qualità, la sicurezza del software può essere valutata complessivamente; tuttavia, in questo caso, le caratteristiche olistiche della qualità e della sicurezza divergono. Ritorniamo alla casa descritta precedentemente. Supponiamo che i potenziali acquirenti siano particolarmente attenti agli aspetti della sicurezza. Essi noterebbero le grandi serrature a chiavistello delle porte di fronte e del retro, i fermi su tutte le finestre e una luce esterna. Tutte queste premesse di sicurezza scoraggerebbero i potenziali intrusi. Se, però, ad un'ulteriore ispezione, un pergolato facilmente scalabile portasse a un altro piano con una porta scorrevole non bloccabile, questa unica vulnerabilità trasformerebbe la casa da sicura ad insicura, dimostrando la natura assoluta della sicurezza.

Con il software questa affermazione di sicurezza può essere anche più ingannevole. Prendiamo, per esempio, un sistema di commercio elettronico che utilizza la criptazione dei dati in transito e sofisticati sistemi di autenticazione. Gli utenti e la gestione potrebbero ritenere sicuramente protette le loro risorse. Ma se una vulnerabilità permettesse ad un hacker di effettuare un acquisto usando l'account di qualcun altro, magari cambiando pochi valori nei cookie di sessione nascosti nel browser di un'ignara vittima, ovviamente, quel sistema non sarebbe sicuro. Sorprendentemente, questa specifica vulnerabilità è stata identificata in migliaia di siti di e-commerce negli anni e ancora oggi è riscontrabile in siti di alto profilo.

Questi esempi illustrano un'altra differenza critica tra la qualità e la sicurezza del software: i problemi di sicurezza sono quasi esclusivamente visibili agli addetti ai lavori, mentre la scarsa qualità è, la maggior parte delle volte, dolorosamente avvertibile da tutti.

1.2.3 Controllo di problemi di qualità e sicurezza

Si possono svolgere attività di vario tipo per identificare problemi di qualità e di sicurezza nel software. Queste includono prevalentemente l'analisi statica del codice sorgente, il test dinamico dell'applicazione prima del suo rilascio tramite monitoraggio delle risorse utilizzate e dei potenziali attacchi subiti in questa fase. I seguenti sono alcuni controlli esplicativi di potenziali problemi di qualità e sicurezza nell'applicazione:

- Il controllo statico di problemi di qualità tramite analisi del codice, di cui un esempio è l'"Unnecessary String Construction"; esso identifica i punti in cui il programma passa un oggetto `java.lang.String` al costruttore `String`, che spreca memoria poiché i due oggetti sono funzionalmente identici.

- Il controllo dinamico di problemi di qualità tramite test dell'applicazione, di cui un esempio è il "Deadlock"; esso identifica i casi in cui il programma può andare in stallo aspettando risorse condivise.
- Il controllo statico di problemi di sicurezza tramite analisi del codice, di cui un esempio è il "Buffer Overflow"; esso identifica i punti in cui i dati in ingresso inseriti dall'utente possono produrre un'operazione che genera un buffer overflow.
- Il controllo dinamico di problemi di sicurezza tramite test dell'applicazione, di cui un esempio è il "Missing Error Handling"; esso identifica le applicazioni che sono mal configurate e che, per questo motivo, possono rilasciare informazioni utili ad un potenziale aggressore.

1.3 Principali vulnerabilità dei Sistemi Informativi dovuti a errori di programmazione

Il 90% delle vulnerabilità nel software deriva da due distinte macro-categorie di errori di programmazione, ovvero da *una poco accorta gestione dell'input utente* e da *controlli erronei o assolutamente assenti durante l'allocazione delle aree di memoria adibite a contenere i dati.* A queste vanno ad aggiungersi le problematiche di gestione delle sessioni utente. Inoltre, una percentuale non indifferente di vulnerabilità nel software deriva dall'assenza di meccanismi crittografici a protezione dei dati scambiati in rete o conservati su disco. Vi è, in aggiunta, un fattore di media entità che, seppur non infici in via diretta la sicurezza di un software o di un sistema, consente ad una minaccia esterna di acquisire informazioni preziose sullo stato dell'applicazione stessa e di ottenere utili spunti per progredire gradualmente verso tecniche di attacco più complesse e sempre più finalizzate all'accesso fraudolento o al trafugamento dei dati. Infine, vi sono anche vulnerabilità correlate al controllo degli accessi.

Il CERT, l'organismo che vigila sulle problematiche di sicurezza nel software, stima che dal terzo trimestre 1995 al 2006 sono state scoperte più di 28.056 vulnerabilità. Il fenomeno colpisce indistintamente sia applicazioni open source che applicazioni proprietarie. Molte delle sviste di programmazione sono attribuite alla scarsa conoscenza, da parte degli sviluppatori, delle principali vulnerabilità che minano il software. Nelle seguenti sottosezioni si cercherà di fornire un'ampia trattazione delle minacce oggi conosciute.

1.3.1 Validazione dell'Input

Spesso lo sviluppatore di software non si pone il problema che gli utenti autorizzati e possessori di una regolare password d'accesso potrebbero non essere gli unici interessati ad interagire con la sua applicazione e dà per scontato che l'input acquisito in ingresso dal programma sarà sempre conforme e pertinente al caso.

Le vulnerabilità di *Input Validation* scaturiscono proprio dall'assenza di controlli o da errori nella gestione dei dati inviati dall'utente e sfociano in una serie di tecniche di attacco differenti, solitamente finalizzate all'esecuzione di comandi remoti o alla visualizzazione di dati importanti, di seguito descritte.

Script, Servlet e CGI

Le problematiche di Input Validation sono comuni a tutti gli ambienti ma trovano la loro espressione massima nelle applicazioni Web. Di seguito vengono riportate le principali vulnerabilità causate dal mancato filtraggio dei dati utente che un aggressore può riscontrare su Web script, Servlet e CGI.

Shell Execution Command

Le problematiche di *Shell Execution Command* rientrano nella sfera delle vulnerabilità più note e più sfruttate di sempre. Se, nella casistica degli Overflow, la vulnerabilità di riferimento è lo Stack Overflow, nelle applicazioni Web è, senza dubbio, lo Shell Execution Command.

Resa celebre nel lontano 1996 da un bug nella storica CGI phf, questa problematica di sicurezza permette ad un aggressore di lanciare da remoto qualsiasi comando con i privilegi di esecuzione dell'applicazione o del Web server. Tale problematica si manifesta quando i parametri acquisiti in input vengono passati all'interprete di shell senza essere filtrati. L'esecuzione di un comando non è spesso possibile in modo diretto (ovvero, semplicemente, specificando ciò che si desidera eseguire), ma viene causata da una precisa condizione. Ad esempio, sui sistemi Unix, è possibile utilizzare il carattere ";" per concatenare più comandi fra loro, mentre in molti altri casi la condizione scatenante può essere causata da caratteri differenti come ritorno a carrello (`\x0a`), new Line (`\x0c`) e NULL byte (`\x00`).

Un esempio di script vulnerabile a Shell Execution Command è riportato in Figura 1.1.

File Inclusion

Le problematiche di *File Inclusion* sono solitamente riscontrabili nelle applicazioni Web. Affermatesi negli ultimi anni con il boom dei linguaggi e delle tecnologie di scripting (ASP, PHP, Python, Perl, etc.) si manifestano quando i parametri passati ad uno script vulnerabile non vengono opportunamente verificati prima di essere utilizzati per includere dei file in determinati punti di un portale.

Le problematiche di File Inclusion si distinguono solitamente in due categorie:

1. *Local File Inclusion*: si manifestano quando un aggressore può soltanto passare come parametri di uno script vulnerabile dei file residenti localmente nel sistema. Il loro contenuto viene, così, visualizzato a video nell'esatto punto del

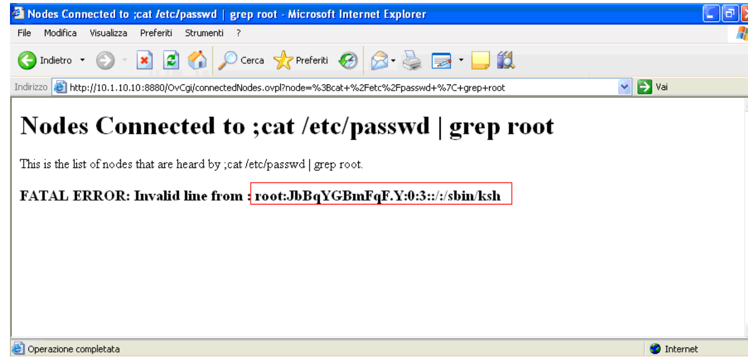


Figura 1.1. Esempio di script vulnerabile a Shell Execution Command

portale in cui si verifica l'inclusione. Un aggressore può, in questo modo, ottenere gli hash delle password di sistema o accedere ad informazioni riservate collocate all'esterno della Document Root del Web Server.
2. *Remote File Inclusion*: è la categoria più pericolosa tra le due perché permette ad un aggressore di passare come parametri di uno script vulnerabile un file che risiede in un altro Web server (ad esempio, un Web server da lui stesso controllato). L'aggressore può collocare all'interno di questo file del codice di scripting (ad esempio, codice PHP malevolo) per eseguire comandi remoti sul sistema.

Le problematiche di Local File Inclusion possono anche essere sfruttate per eseguire comandi remoti se l'aggressore ha la possibilità di collocare localmente un file contenente codice malevolo che può essere puntato dallo script vulnerabile. Il file può essere trasmesso utilizzando i classici servizi di rete (ftp, ssh, cifs, etc.) oppure usufruendo di una qualsiasi procedura di upload richiamabile da Web.

Cross Site Scripting

Il *Cross Site Scripting (XSS)* è una problematica solitamente riscontrabile presso applicazioni Web che consiste nella possibilità di inserire codice HTML o client-side scripting (comunemente Javascript) all'interno di una pagina visualizzata da altri utenti. Un aggressore può, in questo modo, forzare l'esecuzione del codice Javascript all'interno del browser utilizzato dal visitatore.

L'uso più comune del Cross Site Scripting è quello finalizzato all'intercettazione dei cookie e/o dei token di un utente regolarmente autenticato in un portale e, quindi, all'appropriazione indebita delle sessioni Web da esso intraprese. Esistono diverse forme di Cross Site Scripting, ma il funzionamento di base è sempre lo stesso. A variare è, invece, la tecnica utilizzata per forzare l'esecuzione di codice Javascript nel browser del visitatore.

In alcuni casi un aggressore ha la possibilità di iniettare codice persistente nella pagina Web vulnerabile, ovvero codice memorizzato dal server (ad esempio, su un database) e riproposto al client durante ogni singolo collegamento.

In altre circostanze il codice iniettato non viene memorizzato e la sua esecuzione è resa possibile soltanto invogliando l'utente, attraverso tecniche di Social Engineering, a cliccare su un link che punta alla pagina Web vulnerabile. In quest'ultimo caso l'URL viene solitamente rappresentato in formato esadecimale (o altre forme), per evitare che l'utente possa identificare il codice Javascript passato come parametro alla pagina stessa.

In altri casi l'aggressore può beneficiare di tecniche di URL Spoofing per mascherare il codice malevolo.

Le vulnerabilità di Cross Site Scripting (XSS) possono essere, in particolare, sfruttate da un aggressore per:

- prendere il controllo remoto di un browser;
- ottenere un cookie;
- modificare il collegamento ad una pagina;
- redirigere l'utente ad un URI differente dall'originale;
- forzare l'immissione di dati importanti in form non-trusted.

Un esempio di servlet vulnerabile al Cross Site Scripting è riportato in Figura 1.2.

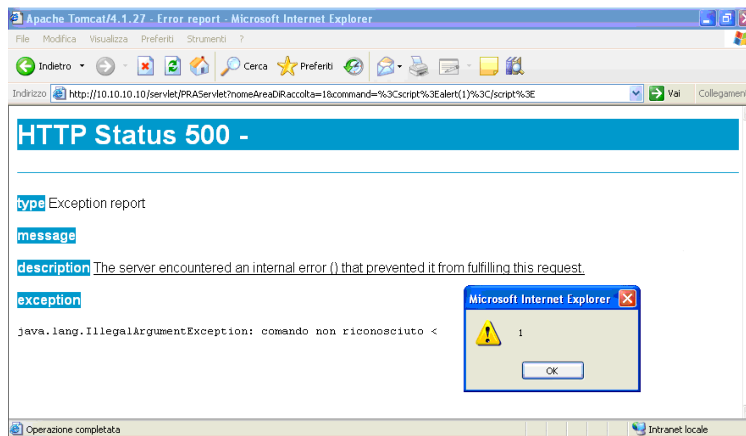


Figura 1.2. Esempio di servlet vulnerabile al Cross Site Scripting

Directory Traversal

Le problematiche di *Directory Traversal*, note anche come *Dot-Dot Vulnerability*, si verificano quando un aggressore ha la possibilità di inviare dell'input che viene utilizzato dall'applicazione per accedere ad un file in lettura e/o scrittura.

Solitamente le applicazioni vietano l'utilizzo di percorsi completi (ad esempio, `/etc/shadow` o `c:\winnt\system32\cmd.exe`) ma, in assenza di controlli sui dati acquisiti in ingresso, un aggressore può ugualmente raggiungere ed acquisire il contenuto di un file residente all'esterno dell'area a lui accessibile, anteponendo una sequenza di punti al nome dello stesso (ad esempio `../../../../nomefile` oppure `.../.../.../nomefile`).

Poiché le problematiche di Directory Traversal sono state utilizzate dagli aggressori fin dallo sviluppo dei primi Web Server, sono oggi tra le più note. Non a caso molte applicazioni vengono progettate in modo da mitigare il rischio del loro sfruttamento.

Alcune fra queste tentano di correggere i dati non validi acquisiti in input trasformandoli in un flusso considerato valido. Comunque, la casistica ha dimostrato che è quasi sempre sconsigliato (al di fuori di specifiche eccezioni) adottare questo approccio in quanto vi è un'alta possibilità di introdurre ulteriori fattori di instabilità o insicurezza all'interno del software sviluppato.

SQL Injection

L'*SQL Injection* è una problematica che colpisce principalmente le applicazioni Web che si interfacciano con un database, anche se non è unicamente circoscrivibile a questo ambito. Infatti, essa è una vulnerabilità in genere molto nota in tutte quelle applicazioni (anche client/server) che interrogano un database. L'SQL Injection si verifica quando uno script, o un'altra componente applicativa, non filtra opportunamente l'input passato dall'utente, rendendo possibile per un aggressore l'alterazione della struttura originaria della query SQL, attraverso l'utilizzo di caratteri speciali (ad esempio, apici e virgolette), oppure mediante la concatenazione di costrutti multipli (ad esempio, utilizzando la keyword SQL UNION).

A seconda delle circostanze e del tipo di database server con cui l'applicazione si interfaccia, l'aggressore può sfruttare una problematica di SQL Injection per:

- aggirare i meccanismi di autenticazione di un portale (ad esempio, forzando il ritorno di condizioni veritiere alle procedure di controllo);
- ricostruire il contenuto di un database (ad esempio, localizzando le tabelle contenenti i token delle sessioni attive, visualizzando le password degli utenti cifrate/non cifrate o altre informazioni di natura critica);
- aggiungere, alterare o rimuovere i dati già presenti nel database;
- eseguire stored-procedure.

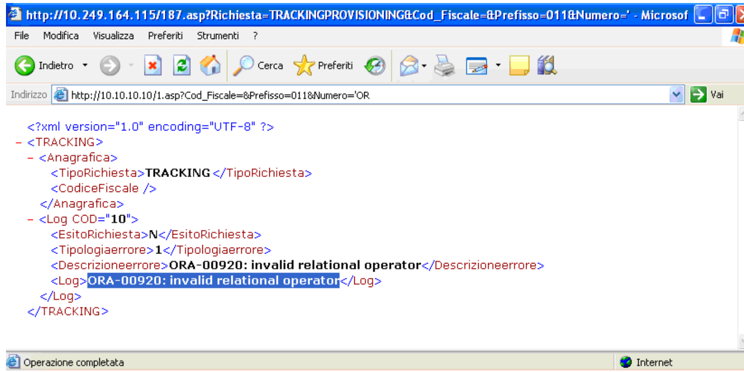


Figura 1.3. Esempio di script vulnerabile a SQL Injection

Un esempio di script vulnerabile ad SQL Injection è illustrato in Figura 1.3.

Le tecniche di SQL Injection sono di varia tipologia; di seguito se ne illustrano due semplici e significative.

Iniezione di una seconda query mediante il carattere ";"

Si consideri la query:

```
$sql = "SELECT * from utenti WHERE id=$id";
```

Se il parametro `$id` fosse acquisito da input ed inizializzato alla stringa:

```
1; DROP table utenti
```

la query risultante sarebbe:

```
SELECT * from utenti WHERE id=1; DROP table utenti
```

che causerebbe la rimozione, da parte dell'aggressore, della tabella `utenti`. Le query multiple non sono, comunque, supportate da tutti i database server.

Autenticazione senza credenziali mediante l'utilizzo degli apici

Si consideri la query:

```
$sql = "SELECT * from utenti WHERE login='$login' AND password='$password'";
```

Se il parametro `$login` fosse acquisito da input ed inizializzato alla stringa:

```
mario
```

e il parametro `$password` fosse acquisito da input ed inizializzato alla stringa:

```
123' OR ''='
```

la query risultante sarebbe:

```
SELECT * from utenti WHERE login='mario' AND password='123' OR ''=''
```

così che l'espressione a destra di `AND` sarebbe sempre vera e, quindi, verrebbe restituita la tupla che ha il campo `login` uguale a `mario` senza l'inserimento della password corretta relativa all'utente mario.

1.3.2 Bound Checking e problematiche di overflow

Le problematiche di overflow si verificano solitamente quando i dati provenienti da input vengono memorizzati all'interno di buffer non abbastanza grandi da contenerli. Ciò causa dei comportamenti differenti.

A seconda delle regioni di memoria in cui l'overflow si è manifestato e delle aree sovrascritte, l'aggressore può eseguire comandi remoti finalizzati all'apertura di un canale di accesso al sistema vulnerabile.

Altre tecniche di overflow si manifestano a seguito di circostanze diverse e non necessariamente correlabili alla copia o allo spostamento di dati in un buffer non capace di contenerli.

Le principali problematiche di Overflow oggi conosciute vengono di seguito descritte.

Stack Overflow

Lo *Stack Overflow* è, per antonomasia, "la tecnica di overflow", nota agli esperti di sicurezza fin dagli anni 80. Il principio è molto semplice e si basa sulla possibilità di saturare un buffer oltre le sue reali capacità di contenimento fino a sovrascrivere l'indirizzo di ritorno della funzione vulnerabile. L'indirizzo di ritorno è un valore posizionato nella regione di memoria Stack che permette all'applicazione, al termine della funzione, di riprendere l'esecuzione del programma dall'istruzione immediatamente successiva alla chiamata della funzione stessa.

Questo valore è puntato da diversi registri in base all'architettura hardware per cui l'applicazione è stata sviluppata (ad esempio, EIP su piattaforma x86 o RIP su piattaforma x64).

Riuscendo a saturare un buffer oltre le sue capacità di contenimento, un aggressore ha la possibilità di sovrascrivere, con valori prettamente arbitrari, tutte le aree di memoria adiacenti, fino a giungere all'indirizzo di ritorno; egli ha, quindi, la possibilità di far proseguire l'esecuzione del programma da qualsiasi indirizzo di memoria desiderato, deviando il regolare flusso esecutivo dell'applicazione.

L'esecuzione di codice malevolo attraverso uno Stack Overflow si sostanzia fondamentalmente in tre passi:

1. l'aggressore satura il buffer non soggetto a bound-checking e colloca lo shellcode ad un certo punto della memoria;
2. l'aggressore sovrascrive l'indirizzo di ritorno della funzione vulnerabile con l'indirizzo in memoria in cui risiede lo shellcode;
3. al termine della funzione viene eseguito lo shellcode.

In Figura 1.4 viene fornita una rappresentazione generica di uno Stack Overflow.

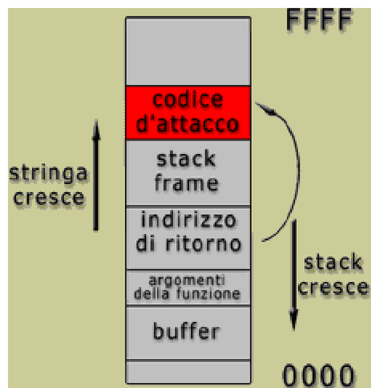


Figura 1.4. Rappresentazione generica di uno Stack Overflow

Off-by-one/Off-by-few

Gli overflow che si manifestano nello stack sono oggi meno frequenti rispetto al passato, ma non sono del tutto scomparsi. In realtà, queste problematiche sono ancora riscontrabili nei moderni software a causa dell'impiego erroneo di funzioni di programmazione considerate sicure. Gli overflow definiti Off-by-one o Off-by-few ne sono la dimostrazione palese.

In questa categoria rientrano tutti gli overflow che, al contrario degli Stack Overflow, permettono di eccedere solo di uno o pochi byte oltre le reali capacità di contenimento di un buffer. Questa condizione, a seconda del compilatore utilizzato, della predisposizione dei buffer e delle variabili in memoria e, quindi, soprattutto, dell'architettura hardware su cui il software gira, può permettere ad un aggressore di alterare a piacimento il flusso di esecuzione dell'applicazione, senza intaccare in modo diretto l'indirizzo di ritorno della funzione vulnerabile.

In genere è sufficiente raggiungere l'ultimo byte dell'indirizzo dello stack frame della funzione vulnerabile (ad esempio, il frame pointer puntato, nell'architettura hardware x86 dal registro EBP) per poter sfruttare l'attacco eseguendo uno shellcode. Questo genere di errori si verifica molto spesso all'interno di cicli.

Format String

Il *Format String Overflow* è una tecnica abbastanza recente dal momento che risale alla metà del 2000. Precedentemente nota per i soli effetti di blocco di un'applicazione, questo genere di overflow si manifesta indistintamente nella regione di memoria Stack o Heap quando una funzione non specifica deliberatamente il formato delle stringhe elaborate e quando questa possibilità viene lasciata all'utente.

In presenza di una funzione vulnerabile, tramite il format string "`%n`", un aggressore può, infatti, scrivere un valore arbitrario in un qualsiasi punto dello spazio di memoria allocato per il processo dell'applicazione.

L'esecuzione di codice malevolo attraverso un Format String Overflow si sostanzia fondamentalmente in tre passi:

1. l'aggressore colloca in un certo punto della memoria lo shellcode;
2. egli individua in memoria l'indirizzo di ritorno della funzione vulnerabile e lo sovrascrive con l'indirizzo in cui risiede lo shellcode;
3. quando termina la funzione verrà eseguito lo shellcode.

Questa tecnica è soggetta a variazioni nel caso di buffer che risiedono nella regione di memoria Heap, dove per eseguire lo shellcode è eventualmente possibile sfruttare indirizzi di chiamata a funzioni di hook, puntatori a funzioni di distruzione (destructor) invocate all'uscita dell'applicazione, puntatori a gestori delle eccezioni oppure puntatori a funzioni residenti in librerie esterne linkate con l'applicazione.

Un aggressore può utilizzare uno di questi puntatori anche nel caso in cui l'overflow si manifesti nella regione di memoria Stack (ad esempio, per bypassare restrizioni di tipo stack canary/cookie o in quelle architetture in cui lo Stack non risulti essere eseguibile).

Heap Overflow

I buffer allocati dinamicamente da un'applicazione risiedono nella regione di memoria Heap e sono sottoposti a problematiche di overflow così come quelli residenti nello Stack. Un luogo comune del passato, ormai sfatato, era che problematiche di questo tipo non potessero essere sfruttate da un aggressore per eseguire uno shellcode per via dell'assenza di un indirizzo di ritorno che potesse essere utilizzato come puntatore al codice malevolo.

Un *Heap Overflow* si manifesta solitamente quando un buffer che viene deallocato contiene dati arbitrari provenienti da input utente o quando, successivamente ad un overflow, ne viene allocato uno nuovo. Entrambi i casi, a seconda dell'architettura, forzano il verificarsi di una specifica condizione e, quindi, l'esecuzione di uno shellcode. La tecnica è resa possibile manipolando i puntatori alle aree di memoria (chunk) che vengono liberati/allocati.

L'esecuzione di codice malevolo attraverso un Heap Overflow si sostanzia fondamentalmente in quattro passi:

1. L'aggressore colloca in un certo punto in memoria lo shellcode e sovrascrive opportunamente il buffer residente nell'Heap.
2. Egli sollecita o attende che l'area di memoria sovrascritta venga liberata dall'applicazione o ne venga sequenzialmente allocata una nuova.
3. In seguito ad uno degli eventi descritti al punto 2, l'indirizzo dello shellcode viene collocato in un punto in memoria arbitrariamente scelto dall'aggressore tramite la manipolazione dei puntatori memorizzati nella struttura che descrive il chunk liberato/allocato. Punti validi sono, ad esempio, indirizzi di chiamata a funzioni di hook o puntatori a gestori delle eccezioni.
4. Lo shellcode viene eseguito.

Sovrascrivere un puntatore a file

Non tutti gli overflow che si manifestano nella regione di memoria Heap possono essere sfruttati per eseguire uno shellcode sul sistema. Ad esempio, quando un Heap Overflow si manifesta in memoria in prossimità di un puntatore ad un file, l'aggressore può alterarlo e sollecitare la scrittura di dati arbitrari in un punto diverso del disco. Un aggressore può, in questo modo, aggiungere al sistema un nuovo utente con password nulla, cambiare da remoto la configurazione di un'applicazione disattivando alcune sue funzionalità di sicurezza o aggiungendovi direttive originariamente non previste.

Integer Overflow ed altri errori logici di programmazione

Inizialmente con il termine *Integer Overflow* si tendeva a descrivere una moltitudine di vulnerabilità differenti tra loro. Solo nel 2002 questo tipo di problematica è stata circoscritta ad una specifica condizione che si verifica quando un'applicazione effettua un'operazione matematica di addizione, sottrazione o moltiplicazione su un intero con segno, acquisendo un operando da input utente, e non considerando i casi in cui il valore numerico ottenuto può essere negativo o minore/maggiore del previsto.

Poiché l'aggressore ha la possibilità di specificare un valore arbitrario, può causare uno Stack o un Heap overflow auto indotto quando il risultato dell'operazione matematica viene utilizzato per specificare la dimensione di un buffer,

forzandone un'allocazione non sufficiente a contenere i dati acquisiti in ingresso dalla funzione vulnerabile.

Una problematica simile si verifica anche nei casi in cui un valore numerico acquisito da input utente viene convertito in un formato differente rispetto alla variabile originaria che lo contiene. A seconda del tipo di conversione il risultato finale può differire notevolmente in eccesso o in difetto dal valore iniziale, causando l'allocazione di buffer insufficienti a soddisfare la necessità di contenimento dei dati o lo spostamento/la copia di un numero di byte eccessivo da un'area di memoria all'altra.

Un terzo fattore di instabilità in un'applicazione può derivare dalla comparazione di interi con segno. Un aggressore può sfruttare questo genere di errori per bypassare i controlli di sicurezza dell'applicazione e giungere alla sollecitazione di una condizione traducibile nell'esecuzione di codice remoto. Più frequentemente gli Integer Overflow o le vulnerabilità derivate da calcoli matematici su interi possono indurre al blocco dell'applicazione o di un suo thread.

1.3.3 Session Management

Le problematiche di *Session Management* sono particolarmente comuni nelle applicazioni Web e, più in generale, in tutte quelle applicazioni in cui ricopre particolare importanza la gestione e la differenziazione delle sessioni di collegamento di ciascun client. In questo caso, errori di progettazione del software possono indurre alla possibilità, per utenti non autorizzati, di accedere a dati protetti; in questo modo, un aggressore può appropriarsi della sessione di collegamento di un utente lecito operando al suo posto mentre un utente regolare potrebbe non poter accedere ad una o più risorse.

Di seguito sono descritte le principali cause e minacce che possono portare a problematiche di Session Management.

Session Stealing ed Hjihacking

Essenzialmente un aggressore che riesce ad ottenere l'identificativo della sessione (detto anche token) o il cookie di un utente, e riesce a replicarlo esattamente in una o più richieste inviate al server ha la capacità di accedere ad aree o risorse che dovrebbero essere riservate soltanto all'utenza lecita, aggirando in modo diretto i meccanismi di autenticazione dell'applicazione.

Diverse sono le cause che agevolano o permettono di portare a termine attività di *Session Stealing/Session Hjhacking*. Le più comuni vengono riportate di seguito.

Cookie

L'attacco attraverso il quale un aggressore riesce solitamente ad appropriarsi in modo indebito del cookie di un altro utente è il Cross Site Scripting. Altri fattori

in fase di sviluppo dell'applicazione influenzano comunque la possibilità di portare a termine con successo un'attività di Session Stealing. Questi, in particolare, sono:

- la generazione di cookie il cui tempo di scadenza non è chiaramente indicato;
- la generazione di cookie persistenti nel disco del client anche dopo il termine della sessione;
- la generazione di cookie non cifrati e trasmessi tramite richieste clear-text;
- la validità del cookie anche dopo un periodo di inattività dell'utente molto lungo;
- l'assenza dell'attributo HttpOnly in fase di generazione del cookie, che ne agevola l'accesso a script client-side;
- l'utilizzo di valori ricorrenti e non casuali che compongono il cookie durante la sua generazione.

Token di sessione

Un token è un identificativo che correla univocamente una sessione ad un utente. Esso, una volta generato, viene collocato all'interno del cookie oppure viene propagato attraverso l'URL per fare in modo che l'applicazione riconosca con esattezza l'utenza e determini, in base ai suoi privilegi, le azioni che può o meno svolgere sul portale.

Un aggressore può appropriarsi di un token di sessione in almeno tre modi:

1. creandolo sul momento (ad esempio, quando il meccanismo di generazione del token è banale, non si basa su valori casuali ed è facilmente ricostruibile a partire dal nome dell'utente);
2. forzando l'utente a rivelarlo con un meccanismo di copia e incolla (ad esempio, con tecniche di Social Engineering);
3. indovinandolo attraverso tecniche di Brute Forcing; ciò è possibile quando l'identificativo della sessione viene generato con valori non casuali o utilizzando una bassa entropia.

Accesso ad aree non autorizzate

Un aggressore può, in talune circostanze, disinteressarsi dei cookie o dei token quando è in grado di attivare una nuova sessione con i privilegi dell'utente desiderato; ciò può avvenire:

1. aggirando il normale meccanismo di autenticazione dell'applicazione;
2. facendo leva su alcuni errori logici dell'applicazione per ottenere la password corrente o sollecitarne un cambio;
3. praticando brute forcing della password direttamente dal modulo di autenticazione dell'applicazione.

Nel primo caso l'aggressore può sfruttare problematiche di Directory Listing o Directory Traversal per accedere ad aree dell'applicazione che dovrebbero essere visibili solo previa autenticazione.

Il secondo caso si manifesta, invece, solitamente quando:

- la procedura di reset della password dell'applicazione fallisce nell'inviare la password all'utente corretto oppure permette all'aggressore di cambiare impropriamente la casella e-mail in cui la stessa viene trasmessa;
- la password è facilmente determinabile a partire dalla risposta che può essere fornita alla domanda posta per ricordarla (nel caso in cui sia questo il meccanismo di recupero adottato);
- le password di accesso possono essere recuperate in forma cifrata o in testo chiaro dal file system o dal database sfruttando problematiche di Directory Listing, Directory Traversal, SQL Injection, etc.

Nel terzo caso l'aggressore può, invece, di proposito o involontariamente, determinare il blocco dell'account utente a causa dei meccanismi di lock-out che potrebbero scattare quando l'applicazione rileva un certo numero di tentativi di login falliti. Questo genere di interventi è classificabile nella categoria degli attacchi DoS.

1.3.4 Crittografia

Con il termine *crittografia* si indica una metodologia adottata fin dagli albori della civiltà umana per rendere un messaggio comprensibile solo alle parti autorizzate a leggerlo. Applicata prevalentemente in ambito militare, la crittografia rappresenta oggi uno degli strumenti più proficui per sviluppare applicazioni software sicure, capaci di rispondere alle necessità crescenti di garanzia dell'integrità e della riservatezza dei dati.

Di seguito vengono riportare le tecniche più comunemente utilizzate dagli aggressori per appropriarsi in modo fraudolento di informazioni private, invertendo il loro processo di cifratura, e le vulnerabilità più comuni che permettono il verificarsi di tali condizioni.

Sniffing ed algoritmi crittografici deboli

Uno dei principali motivi addotti a favore dell'uso della crittografia è quello di preservare la riservatezza dei dati che vengono scambiati in rete. Le applicazioni che non implementano alcun meccanismo crittografico sono le più esposte a tecniche di *Sniffing*.

L' aggressore che riesce ad attestarsi in un punto qualsiasi fra i due nodi che comunicano (ad esempio, nel gateway d'uscita del server) o che riesce a forzare il redirect del traffico verso la sua postazione, può, in pratica, determinare con

estrema semplicità il corretto contenuto delle sessioni applicative, intercettando e ricostruendo il flusso dei dati in testo chiaro. In questo caso, per appropriarsi delle informazioni trasmesse, non deve portare a termine nessuno sforzo e nessuna procedura di decrypting.

Cifrare i dati può non rappresentare la risoluzione al problema dello Sniffing. In questo contesto gioca, infatti, un ruolo fondamentale il tipo di algoritmo implementato dall'applicazione e la dimensione della chiave di cifratura utilizzata. Pur in presenza di sessioni cifrate un aggressore può, infatti, intercettare ed archiviare ugualmente tutto il traffico per cercare di decifrarlo in modalità offline, ovvero a sessione client/server terminata.

Se l'applicazione implementa un algoritmo semplice e/o fa uso di una chiave crittografica di dimensioni non adeguate, un aggressore può, eventualmente, riuscire a decifrare i dati scambiati anche in tempo reale. Le principali tecniche utilizzate per rompere una chiave crittografica generata attraverso algoritmi simmetrici o di hashing sono il Brute Forcing e la Rainbow Table; esse vengono descritte in dettaglio nelle prossime sottosezioni.

Brute Forcing

Il *Brute Forcing* è la tecnica principalmente utilizzata da un aggressore per "rompere" la chiave crittografica di un messaggio testuale o di una sequenza di byte cifrata (ad esempio, una password).

Non unicamente circoscrivibile all'ambito crittografico (un attacco Brute Forcing può, tra gli altri casi, anche palesarsi tramite tentativi multipli di accesso ad un servizio utilizzando una lista di username o di password già predefinite); tale tecnica consiste essenzialmente nel tentare in modo sistematico tutte le possibili combinazioni di un valore crittografato, cifrando ciascuna combinazione prodotta con lo stesso algoritmo utilizzato per proteggere tale valore crittografato e confrontando il risultato con il ciphertext originale.

Un eventuale match identifica la chiave che può essere impiegata per riportare l'intero messaggio o la sequenza di byte in formato clear-text.

Il Brute Forcing è una tecnica che, a seconda dell'algoritmo crittografico utilizzato per cifrare un messaggio e soprattutto della dimensione della chiave, può non raggiungere l'intento di un aggressore in tempi ragionevoli.

Tale tecnica viene solitamente sfruttata per decifrare password o chiavi cifrate con algoritmi simmetrici. Nel seguito esamineremo le principali modalità con cui si può esplicitare una tecnica di Brute Forcing.

Weak Keys

Il processo di brute forcing può essere totalmente scartato dall'aggressore se il meccanismo di generazione automatico delle chiavi crittografiche di un'applicazione produce delle *Weak Keys*. Si tratta di chiavi che, quando utilizzate per cifrare un messaggio, generano in output lo stesso messaggio in testo chiaro.

Questa problematica è strettamente correlata al tipo di algoritmo crittografico utilizzato e può essere comunemente riscontrata durante la generazione di chiavi DES, 3DES, RC4, Blowfish, IDEA, etc.

Collisioni

Lo stesso concetto delle Weak Key per gli algoritmi crittografici simmetrici è applicabile, in modo un po' diverso, per gli algoritmi di hashing one-way (MD5, SHA-1, etc.). Quando un'applicazione utilizza questo genere di algoritmi, ad esempio per confrontare la password fornita da un utente con quella presente in un database, il valore in testo chiaro proveniente da input viene convertito in hash (una stringa cifrata). L'hash viene, successivamente, confrontato direttamente con il valore, sempre cifrato, mantenuto nel database.

Per alcuni algoritmi (come MD5) è matematicamente dimostrato che è possibile cifrare valori testuali diversi che producono in output lo stesso hash. Questa condizione, definita appunto *Collisione*, può essere utilizzata da un aggressore per autenticarsi in un portale fornendo delle credenziali di accesso differenti da quelle originali.

Rainbow Table e Salt Value

Una *Rainbow Table* è, concettualmente, una tabella in cui viene mantenuto un numero cospicuo di hash per i quali è già conosciuto il valore originario in testo chiaro. Semplicemente inserendo un hash in un software che implementa il concetto di Rainbow Table, un aggressore può, quindi, determinare in pochi secondi l'esatta corrispondenza in testo chiaro.

Questa problematica si verifica principalmente quando l'applicazione, per generare un hash, non utilizza un salt value, ovvero un fattore casuale che modifica la conformazione in output dell'hash stesso e non permette di utilizzare le classiche Rainbow Table per la relativa conversione in testo chiaro.

Archiviazione insicura

La trasmissione attraverso la rete di dati in testo chiaro, oppure cifrati con algoritmi crittografici deboli, non è l'unica pratica che può portare alla loro appropriazione indebita da parte di un aggressore. Anche archiviarli allo stesso modo nel file system o in un database può sfociare nel loro trafugamento o nella loro alterazione. Sfruttando la combinazione di una o più problematiche di:

- Input Validation (ad esempio, Directory Traversal, SQL Injection, etc.);
- Buffer Overflow;
- Error e Time Handling (ad esempio, Directory Listing, etc.).

un aggressore può, infatti, ottenere accesso a questi dati da un'applicazione di front-end oppure si può introdurre direttamente nel sistema che li ospita. Dall'interno, attraverso tecniche di File System Polling, egli ha, inoltre, maggiori possibilità di alterare o visualizzare i dati per i propri scopi, anche quando vengono cifrati a seguito di un processo di pre-elaborazione.

File System Polling

Non direttamente correlabile con problematiche crittografiche in senso stretto, questa tecnica viene spesso utilizzata da un aggressore con accesso locale ad un sistema per appropriarsi dei dati fintanto che essi permangono in forma non cifrata nel disco. Questa condizione si verifica quando tali dati vengono collocati per l'elaborazione e per lunghi periodi in tabelle di staging o in punti del file system sempre uguali, prima di essere definitivamente cifrati.

L'aggressore, utilizzando script automatici, può, in questo modo, copiare ciclicamente il contenuto di queste tabelle e directory in locazioni del disco differenti e mantenere i relativi dati in forma intellegibile per un periodo di tempo sufficientemente lungo a garantirne la visualizzazione o la successiva modifica.

1.3.5 Error e Time Handling

La gestione degli errori, delle eccezioni o delle circostanze fuori dalla norma sono tutti quanti aspetti frequentemente bistrattati dagli sviluppatori software che possono indurre l'applicazione a:

1. bloccarsi o sospendersi;
2. rilasciare informazioni utili all'aggressore per avanzare con successo nella sua azione intrusiva nel sistema;
3. permettere all'aggressore di acquisire il controllo diretto del sistema o dell'applicazione.

Di seguito vengono trattate le tecniche più comuni che possono causare l'insorgere delle problematiche descritte nei punti precedenti.

User Enumeration

Le problematiche di *User Enumeration* si manifestano su quei servizi o quelle applicazioni che non gestiscono opportunamente le condizioni di errore durante le fasi di login e/o interrogazione, ritornando messaggi specifici e non generici.

Esse colpiscono prevalentemente i portali Web, seppur l'ambito di sfruttamento non sia unicamente circoscrivibile a questo genere di ambienti.

Le applicazioni o i servizi soggetti a tale problematica vengono stressati da un aggressore con apposite richieste. In base alle risposte ottenute, egli è in grado di determinare le utenze valide o quelle inesistenti nel sistema/portale.

La possibilità di determinare gli utenti regolari permette ad un aggressore di utilizzare le informazioni acquisiste come base di partenza per attacchi intrusivi più precisi e mirati.

Ad esempio, se, a seguito di un processo di autenticazione, l'aggressore ottenesse in risposta alla sua richiesta di login il messaggio specifico "Nome Utente Errato" avrebbe determinato che l'utenza utilizzata non esiste; viceversa, se la risposta ritornata fosse "Password Errata", avrebbe determinato invece la sua esistenza.

Condizioni simili possono essere riscontrate non solo nei processi di autenticazione, ma anche in quelli di registrazione di un nuovo utente, di recupero password o in applicazioni server per lo scambio di posta elettronica.

Information Disclosure

Le problematiche di *Information Disclosure* sono molto comuni nelle applicazioni Web anche se non unicamente circoscrivibili a questo ambito. Esse si manifestano quando un aggressore riesce, con apposite richieste, a sollecitare una condizione non prevista o mal gestita dall'applicazione; quest'ultima restituisce messaggi informativi o di errore contenenti dati o informazioni che possono agevolarlo durante i suoi attacchi intrusivi.

Non tutte le condizioni di Information Disclosure sono causate da richieste o eventi non correttamente gestiti dall'applicazione. Alla radice di problematiche simili possono anche esservi script o componenti mal progettate che, se interrogate opportunamente con richieste regolari, possono fornire all'aggressore degli spunti utili per proseguire nella sua attività intrusiva.

Sono classificabili come derivanti da problematiche di Information Disclosure le seguenti informazioni rilasciate dall'applicazione ad utenze anonime o non autorizzate a seguito di richieste malevole o regolari:

- i dati che svelano il percorso o i percorsi su disco in cui gli script o le componenti dell'applicazione sono state installate e risiedono;
- i dati correlabili allo stato attuale dell'applicazione, alla sua versione ed agli eventuali moduli o plug-in installati;
- i dati correlabili ai log delle attività manutentive svolte sull'applicazione;
- tutti gli altri dati eventualmente svelati che, per l'azienda, hanno valenza critica, personale o sensibile.

Un esempio di script Web soggetto a Information Disclosure viene mostrato in Figura 1.5.

Le applicazioni compilate con l'opzione debugging o verbose possono essere più facilmente soggette a problematiche di Information Disclosure. Molte di queste condizioni si verificano, inoltre, a causa di una poco accorta gestione dell'Input Utente.

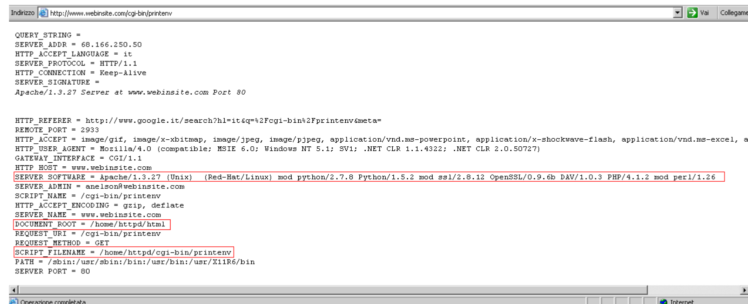


Figura 1.5. Esempio di default script Web soggetto a Information Disclosure

Directory Listing

Le problematiche di *Directory Listing* sono molto comuni nelle applicazioni Web anche se non unicamente circoscrivibili a questo ambito. Esse si manifestano quando un aggressore riesce, con apposite richieste, a visualizzare il contenuto di una directory, prelevando file dal suo interno o visualizzando dati che dovrebbero di norma essere preclusi agli utenti non autenticati o che non dispongono di specifici privilegi.

Comunemente un aggressore riesce a sfruttare questo tipo di problematiche facendo leva su configurazioni applicative errate.

La Figura 1.6 riporta un esempio di Directory Listing.

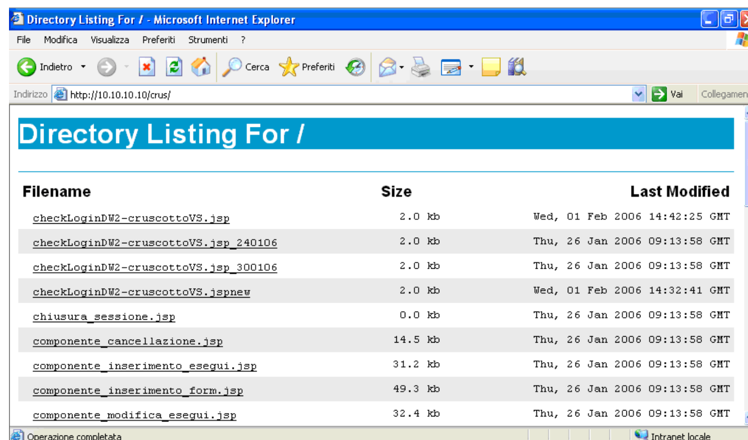


Figura 1.6. Esempio di Directory Listing

Denial Of Service

Traduzione di "negazione del servizio", un *Denial Of Service* è una condizione che causa, a seconda di specifiche circostanze, il blocco o la sospensione dell'applicazione, di un suo singolo processo, di un'unica componente o dell'intero sistema. Ciò è determinato dal tipo di integrazione dell'applicazione stessa con il kernel, dalle sue strutture e dai privilegi con la quale viene eseguita.

Una condizione di Denial Of Service viene comunemente causata da un aggressore che sfrutta errori di programmazione riconducibili a problematiche di overflow o come effetto di un attacco non andato a buon fine, che mirava originariamente all'esecuzione di uno shellcode.

Condizioni di Denial Of Service meno pesanti possono causare il blocco di un account utente.

Deadlock

Nella programmazione multithread uno degli errori più comuni che sfocia in problematiche di Denial Of Service è il *deadlock*. Questa circostanza si verifica quando due o più processi attendono l'un l'altro, a tempo indefinito, il termine di esecuzione di una procedura o il rilascio di una risorsa che causa il blocco del sistema o dell'applicazione.

Race Condition

Una *Race Condition* è una condizione che permette di deviare il flusso di output o il comportamento di un processo applicativo la cui esecuzione è strettamente dipendente da precise sequenze procedurali o da eventi correlabili con il tempo. Essa si verifica quando l'accesso multiplo alle risorse (file, dispositivi di rete, variabili, etc.) non viene opportunamente controllato.

La circostanza più classica in cui si presenta una Race Condition è riconducibile a quelle applicazioni che devono scrivere dei dati sul disco e, prima di eseguire tale operazione, procedono ad una serie di controlli preventivi. Un aggressore può usufruire del lasso di tempo in cui questi controlli vengono effettuati oppure può bloccare per un sufficiente periodo la loro esecuzione sfruttando una vulnerabilità logica della stessa applicazione (ad esempio, un deadlock momentaneo), per alterare il collegamento o l'associazione del file che deve essere acceduto ad una diversa destinazione del disco e per forzare la scrittura di dati arbitrari.

Ad esempio, se l'applicazione vulnerabile viene eseguita con i privilegi di amministratore e l'aggressore possiede i permessi di un normale utente di sistema, una condizione di Race Condition può permettergli di alterare il contenuto dei file di cui root è proprietario (ad esempio, quelli in cui vengono mantenute le password di sistema o le dichiarazioni dei gruppi) pur non possedendo i necessari privilegi per portare a termine l'operazione.

Privilege Escalation e Bypassing dei permessi utente

Le eccezioni e le condizioni non previste o mal gestite da un'applicazione determinano, nella maggior parte delle circostanze e nei soli casi di attacchi andati a buon fine, una situazione di *Privilege Escalation* per l'aggressore, ovvero la possibilità di svolgere operazioni sul sistema o sulla stessa applicazione con privilegi superiori rispetto a quelli posseduti originariamente prima dell'attacco.

Ad esempio, sfruttando con successo uno Stack Overflow, l'aggressore che, da remoto, poteva unicamente godere dei privilegi riservati ad un utente anonimo o di basso profilo, può successivamente operare nel sistema come se fosse un utente locale a cui sono stati assegnati permessi di amministratore.

Allo stesso modo, nel caso di una problematica di tipo Race Condition, l'aggressore può modificare un file pur non possedendo, come utente originario, gli effettivi privilegi di scrittura. Nel caso di un Directory Listing può, invece, accedere ad aree riservate di un portale ancor prima di autenticarsi, aggirando il meccanismo con il quale l'applicazione assegna i permessi agli utenti regolari.

Le motivazioni che rendono solitamente possibile una scalata dei privilegi sono di seguito menzionate:

1. il servizio, la componente o l'applicazione vengono avviati con i privilegi di amministratore;
2. l'applicazione non procede al cleaning dei privilegi di amministratore eventualmente posseduti quando svolge azioni per conto di un utente non privilegiato;
3. nei sistemi Unix o derivati il bit Set-User-ID è attivo.

Un attacco di Privilege Escalation non si definisce tale solo quando l'innalzamento dei privilegi riguarda direttamente il passaggio da un utente non privilegiato ad uno privilegiato, ma anche quando lo scambio di permessi avviene tra utenti non privilegiati.

1.3.6 Processi di Tracciamento

Il tracciamento delle operazioni svolte dagli utenti è una delle attività più critiche per un'applicazione perché l'implementazione di un meccanismo di logging erroneo permette ad un aggressore di mascherare le sue operazioni, di sospendere il servizio o, in taluni casi, di eseguire comandi remoti sul sistema che ospita l'applicazione vulnerabile.

Errori comuni nei meccanismi di tracciamento

Gli errori che permettono ad un aggressore di oscurare le sue operazioni, di sospendere il servizio di tracciamento dell'applicazione o, in talune circostanze, di eseguire codice remoto, vengono riportati di seguito.

Agevolazione delle attività malevole dell'aggressore

Una delle principali preoccupazioni di un aggressore che sferra o porta a termine un attacco a fini intrusivi è di rimuovere ogni traccia delle sue attività per non essere chiaramente identificato. Se questa opportunità gli viene data a priori, egli ha la possibilità di nascondere, anche senza ottenere accesso locale al sistema, quelle operazioni che non sono andate a buon fine. Il meccanismo di tracciamento non fornirà, così, all'amministratore del sistema o del servizio alcuna evidenza dell'attacco subito e non gli permetterà di implementare alcuna misura di contrasto. Le cause più comunemente riconducibili a questo problema derivano:

- da errori nella progettazione del meccanismo di tracciamento dell'applicazione che fallisce nel registrare specifiche attività svolte dagli utenti, memorizzando su file di log solo alcune delle operazioni svolte (ad esempio, l'autenticazione di un utente, ma non la modifica di una particolare risorsa);
- l'impossibilità, da parte dell'amministratore, di configurare modularmente il livello di tracciamento dell'applicazione;
- un livello di tracciamento attivo di default molto basso;
- la presenza di informazioni di natura critica (ad esempio, password di accesso dell'applicazione non cifrate) registrate all'interno dei file di log, congiuntamente a problematiche di Directory Listing o Directory Traversal.

Sospensione del servizio o accesso fraudolento

Se una o più problematiche di Overflow affligono il meccanismo di tracciamento dell'applicazione, ogni tentativo di attacco fallito da parte di un'aggressore può determinare la sospensione del servizio, causando un effetto comparabile a quello di un Denial Of Service. Viceversa, il corretto sfruttamento della tecnica utilizzata potrebbe portare all'esecuzione di comandi remoti nel sistema.

Oscuramento delle attività dell'aggressore

Come descritto precedentemente, tra le principali preoccupazioni di un aggressore vi è quella di oscurare tutte le attività compromettenti e i tentativi di intrusione da lui effettuati. Il metodo più diretto per farlo è quello di ottenere un accesso remoto al sistema e di rimuovere manualmente le tracce lasciate nei file di log. In altri casi è possibile sovvertire direttamente il meccanismo di tracciamento dell'applicazione.

Il filtraggio erroneo di caratteri di controllo ("`\r`“, "`\n`“ o "`\t`“) può, infatti, determinare la registrazione parziale sui file di log delle attività o dei dati di provenienza dell'aggressore (indirizzo IP, utenza utilizzata per condurre la frode, tipo di operazione svolta, etc.).

1.4 Principi di Sicurezza per il Codice Sorgente

La *sicurezza applicativa* è un concetto differente dalla *sicurezza della rete* e *dei sistemi*, ma altrettanto importante. Tipicamente un software fornisce accesso a dati e risorse; quindi un'applicazione insicura può rappresentare un ponte verso l'Enterprise Network ed è, pertanto, necessario difenderla e renderla immune dalle minacce provenienti sia dall'esterno che dall'interno di un'azienda.

1.4.1 Origine delle Vulnerabilità

Comunemente si pensa che le vulnerabilità dei software siano dovute all'incapacità e/o all'indolenza dei programmatori. In realtà, i programmatori hanno tutta l'intenzione di scrivere un buon software esente da difetti di sicurezza. Nonostante un programma soddisfi rigide specifiche funzionali, conterrà immancabilmente vulnerabilità di varia natura; esse sono da imputare a persone che hanno fatto il loro meglio e che, comunque, si ritengono soddisfatte del loro lavoro svolto. È naturale chiedersi perché sia così difficile scrivere codice sicuro e come sia possibile che esistano vulnerabilità che perdurano addirittura per decenni.

Vi sono tre gruppi di fattori avversi alla scrittura di codice sicuro. Essi sono di natura tecnica, psicologica ed economico-sociale [6].

Fattori Tecnici

Produrre un software sicuro è intrinsecamente difficile. Le applicazioni sono spesso costituite da varie componenti separate, ciascuna delle quali, presa singolarmente, può essere considerata perfettamente sicura. Tuttavia, quando queste componenti sono messe insieme, possono creare una falla che potrebbe essere sfruttata.

Non è l'eccessiva complessità di un singolo algoritmo a generare errori, in quanto un programmatore esperto può analizzarlo completamente. I problemi di sicurezza sono strettamente connessi alla straordinaria complessità intrinseca dei calcolatori elettronici.

Fattori Psicologici

I programmatori sono esseri umani. Questo è un fatto che, molto spesso, sembra sfuggire agli analisti di sicurezza quando esaminano le cause delle vulnerabilità. Tutti conveniamo sul fatto che "errare è umano", eppure ci si lamenta spesso della fallibilità degli ingegneri del software. È molto difficile per gli esseri umani essere inclini a particolari generi di valutazione del rischio, come quello di determinare quanto sia difficile proteggere le password agli occhi dei ficcanaso nella rete.

Oltretutto, immaginare esattamente il modo secondo cui un calcolatore esegue un pezzo di codice è una attività mentale piuttosto complessa.

Accanto a questi ci sono, però, anche altri fattori da considerare. Ai fini della sicurezza è, senza dubbio, utile immedesimarsi nella mentalità di un attaccante. Egli, per scovare falle nella sicurezza, pensa come un alieno: osserva ogni cosa come se fosse la prima volta, in modo grezzo, senza alcun contesto socializzato.

D'altra parte, la natura del software può essere vista da vari punti di vista:

- una serie di algoritmi astratti;
- linee di testo su un foglio di carta o uno schermo;
- una serie di istruzioni per un particolare processore;
- un flusso di zeri ed uno nella memoria del computer, oppure memorizzati in supporti magnetici o ottici;
- una serie di librerie concatenate di procedure, codici di terze parti e applicativi originali;
- un flusso di segnali ottici ed elettronici attraverso percorsi elettromeccanici di vario tipo.

In modo meno lineare un software può anche essere visto come:

- una serie di strati "verticali" (il trasporto, il protocollo e la rappresentazione);
- una serie di stadi "orizzontali" (un firewall, un'interfaccia grafica utente, una business logic e un database server);
- una serie di eventi che avvengono in un determinato lasso di tempo ed in un ordine controllato;
- un coinvolgimento di varie locazioni: quando viene eseguita un'applicazione, dov'è l'utente? Dov'è il codice stesso? Dov'è l'host? Dov'è il server? Dov'è il database?

Fattori Economici e Sociali

Tuttoggi, gran parte del software è scritto da persone che non hanno alcuna preparazione in ingegneria del software pregiudicando, quindi, anche l'aspetto della sicurezza. Ciò nonostante, anche quando gli applicativi vengono sviluppati all'interno di prestigiose software house, i programmatori sono sottoposti a pressione; per poter far fronte alla particolare fluidità delle offerte dettata dal mercato e dalla concorrenza occorre rispettare precise tabelle di marcia. Quindi, questioni di carattere economico impongono ridotti time-to-market e, pertanto, tempistiche di realizzo e test molto brevi. Cosicché si perviene a compromessi e si punta ad un livello di sicurezza "ragionevole".

Con il tempo si sono affermate sempre più frequentemente idee sbagliate, ma comunemente accettate fra gli sviluppatori, che hanno portato alla creazione ed al rilascio di applicazioni insicure; alcune di queste idee sono di seguito riportate:

- *se l'applicazione non dà problemi è sicura*: questo è vero soltanto finché essa non viene compromessa;

- *gli errori di runtime non sono importanti*: questo non è assolutamente vero in quanto spesso espongono all'esterno importanti informazioni e sprecano risorse;
- *la soluzione è il penetration testing*: è piuttosto raro che vengano effettuate sessioni di penetration testing complete ed esaustive; ciò perché tali sessioni sono molto complesse e, per essere portate a termine, richiedono esose risorse e lunghe tempistiche;
- *i Web service non sono vulnerabili*: in genere sono i componenti meno testati dal punto di vista della sicurezza.

1.4.2 Importanza di una Programmazione Sicura

Secondo l'istituto di ricerca *Gartner*, il 75% degli incidenti di sicurezza avvengono a livello applicativo; per il *NIST* (National Institute of Standards and Technology), il 92% delle vulnerabilità risiedono a livello applicativo; secondo *Theo de Raadt* (responsabile e sviluppatore OpenBSD), il 95% dei problemi di sicurezza dipendono da errori di programmazione a basso livello.

Tutte queste affermazioni confermano quanto dichiarato dagli analisti Gartner, e cioè che, rispetto agli incidenti di sicurezza, "... i programmatori hanno una responsabilità tre volte superiore a quella degli amministratori di sistema." È, dunque, oramai chiaro a molte aziende che è divenuto fondamentale:

- proteggere i dati sensibili;
- assicurare l'integrità delle informazioni;
- implementare corrette politiche di autenticazione ed autorizzazione;
- fare in modo che il codice sorgente scritto sia conforme con le principali raccomandazioni internazionali.

1.4.3 Progettazione e Sviluppo in Sicurezza dell'Applicazione

Affinché le linee di progettazione e sviluppo rendano sicura un'applicazione è necessario che vengano rispettate delle direttive; le più importanti sono di seguito riportate.

Linee Guida Generiche

Privilegi esecutivi minimi

L'applicazione deve essere sviluppata in modo da minimizzare (o da annullare completamente) la richiesta di privilegi di amministratore per l'esecuzione delle singole componenti di cui consta.

Utilizzo di funzioni di gestione delle stringhe

Tutto l'input processato dall'applicazione deve passare per funzioni sicure di gestione delle stringhe che ne prevedano il bound-checking. L'applicazione deve risultare immune da problematiche di tipo stack overflow, off-by-one/off-by-few overflow o heap overflow.

Specifica del formato delle stringhe

Nel codice sorgente di un'applicazione il formato delle stringhe deve essere sempre specificato nei parametri delle funzioni che lo prevedono e non deve essere mai dato per assunto. L'applicazione deve risultare immune da problematiche di tipo format string overflow.

Casting e variabili numeriche

L'input deve essere filtrato in modo che alle variabili o strutture dati interne dell'applicazione non sia possibile assegnare valori negativi (ad esempio, dichiarando array come signed integer), ad eccezione dei casi previsti e per i quali sia stata pianificata la gestione.

In fase di comparazione di due variabili numeriche, dove il contenuto di almeno di una derivi da input, il casting o l'assegnazione di un valore da una variabile all'altra deve avvenire in base alla stessa tipologia (ad esempio, assegnare un valore intero ad una variabile di tipo short è un errore).

L'applicazione deve risultare immune da problematiche di tipo integer overflow, cambi di segno, troncamento di valori numerici, o altri errori di programmazione logico-computazionali.

Input data validation

L'applicazione deve assicurare, attraverso appositi meccanismi di convalida, che tutti i parametri in input specificati dall'utente siano congruenti. In particolare, sui dati acquisiti in ingresso, l'applicazione deve implementare procedure di controllo che limitino il set di caratteri o i valori che possono essere specificati dall'utente, soltanto a quelli rilevanti ai campi o ai moduli di inserimento di pertinenza, al fine di identificare ed annullare gli effetti dei seguenti errori:

- valori out-of-range o non pertinenti (ad esempio, l'immissione di caratteri non numerici nel campo "Anno di nascita");
- caratteri invalidi negli stream o nei data field;
- dati mancanti o incompleti;
- limite del minimo volume di dati richiesto non soddisfatto o del massimo volume di dati acquisibile in ingresso raggiunto.

I caratteri che devono essere filtrati e considerati invalidi dall'applicazione sono i seguenti:

```
; | ! & x20 x00 x04 x0a x0d x1b x08 x7f ~ ' " - * % ` \ / < > ?
$ @ : ( ) [ ] { } .
```

La convalida dell'input utente non deve mai essere svolta lato client.

Fattore di integrità

Il design e l'implementazione dell'applicazione devono assicurare che tutti gli errori e le eccezioni durante il processo e l'elaborazione dei dati acquisiti in ingresso siano correttamente gestiti e non causino il danneggiamento o la perdita di integrità delle informazioni conservate e mantenute dall'applicazione stessa.

Assenza di codice malevolo

L'applicazione sviluppata non deve includere al suo interno alcun tipo di backdoor amministrativa, trojan horse, spyware o, più in generale, alcuna componente malware.

Gestione dell'output

L'applicazione deve fornire in output soltanto le informazioni rilevanti all'uso delle richieste avanzate dagli utenti, rendendo vana la possibilità di qualsiasi tipo di information gathering o disclosure non autorizzato.

Autenticazione, Autorizzazione e Gestione degli Accessi

L'applicazione deve implementare un meccanismo di controllo degli accessi adeguato. Tutte le operazioni svolte dagli utenti e le attività di autorizzazione ed assegnazione dei permessi devono essere subordinate alla politica standard "*Ogni azione è negata se non espressamente consentita*".

Assegnazione dei privilegi utente

L'applicazione non deve assegnare alcun privilegio/permesso all'utente fin quando il processo di autenticazione ed autorizzazione non è stato completato.

Procedura di accesso dell'applicazione

La procedura di accesso e log-on dell'applicazione deve ridurre al minimo le informazioni fornite agli utenti non ancora autenticati. Essa, inoltre, deve attenersi alle seguenti linee guida:

1. non deve fornire con messaggi specifici alcun tipo di aiuto né rendere comprensibile se il processo di autenticazione è fallito a causa del nome utente o della password errata;
2. non deve fornire alcuna chiara indicazione sui ruoli e sui permessi assegnati ad un utente fin quando il processo di autenticazione non viene completato;
3. deve visualizzare un messaggio di avviso sulle sanzioni derivate dall'accesso fraudolento all'applicazione;
4. deve prevedere il mascheramento della password digitata dall'utente non rendendola visibile o nascondendola attraverso simboli (ad esempio, con asterischi);
5. non deve trasmettere la password in testo chiaro nella rete;
6. deve "processare" le informazioni fornite dall'utente per l'accesso solo quando sono complete;
7. deve prevedere procedure configurabili di blocco momentaneo dell'account dopo una serie di tentativi di accesso infruttuosi;
8. deve visualizzare, al termine della fase di autenticazione, la data, l'ora e le informazioni sull'ultimo sistema (indirizzo IP/FQDN) che ha completato con successo la fase di log-on per l'utente che si è autenticato;
9. deve visualizzare nella console dell'amministratore o nei file di log i dettagli di tutti i precedenti tentativi infruttuosi di accesso per uno specifico utente;
10. l'autenticazione non deve mai essere un processo convalidato dal lato del client.

Account standard

L'applicazione non deve essere rilasciata da chi la sviluppa con account utente standard di tipo amministrativo/operativo o con account protetti tramite password di default.

Autorizzazione

L'applicazione deve sempre operare un controllo sui reali privilegi d'accesso dell'utente prima di autorizzare qualsiasi operazione in lettura, scrittura, rimozione o esecuzione. Essa non deve mai essere un processo convalidato dal lato del client.

Generazione dei token

I token dell'applicazione devono essere generati utilizzando algoritmi true random e devono essere analizzati ogniqualvolta l'utente richiede l'autorizzazione a svolgere una qualsiasi azione, al fine di determinarne permessi e privilegi.

Generazione dei cookie

Nelle applicazioni Web, i cookie delle sessioni applicative devono essere cifrati, non persistenti, devono avere il flag secure attivato e l'attributo HttpOnly impostato.

Contenuto del cookie

Un cookie non deve contenere informazioni critiche, quali password, o essere composto da parti predicibili, come username o valori elaborati basandosi su algoritmi sequenziali. L'identificatore della sessione nel cookie deve avere un'entropia pari almeno a 128 bit.

Scadenza dei cookie

Nelle applicazioni Web, ciascun cookie generato deve essere soggetto ad un tempo di scadenza oltre il quale non deve più essere considerato valido.

Logout utente

Quando un utente ha effettuato il log-out, la sessione relativa deve essere invalidata sia sul server (sganciandola nella entry table delle sessioni attive) che sul client (ad esempio, rimuovendo il cookie o svuotando il suo contenuto).

Timeout di sessione

L'applicazione deve prevedere il rilascio della sessione utente dopo un certo periodo configurabile di inattività della sessione stessa.

Isolamento delle funzioni dell'applicazione

È vietata l'implementazione della sicurezza attraverso l'oscuramento delle funzioni a livello di presentazione. È obbligatorio, invece, isolare e rendere inutilizzabili le funzioni che non devono essere rese accessibili agli utenti, direttamente a livello logico (ad esempio, imponendo la consultazione del token della sessione per determinarne i reali privilegi di esecuzione).

Crittografia

Gestione di password, chiavi e certificati

Le password mantenute dall'applicazione o le chiavi private dei certificati non devono essere conservate in forma non cifrata. Le informazioni sulle password e le chiavi devono risiedere in container (aree del file system, tabelle del database, etc.) differenti rispetto ai dati dell'applicazione.

Trasmissione delle password in rete

Le password trasmesse al/dall'applicazione attraverso la rete devono essere cifrate tramite algoritmi simmetrici con chiave pari almeno a 192 bit (inclusi i bit di parità) e, quando possibile, rafforzate con algoritmi di hashing (ad esempio, MD5/SHA1/SHA-2).

Generazione/Conservazione delle password nel file system o nel database

Le password memorizzate nel file system o nel database sotto forma di hash (ad esempio, MD5/SHA-1 etc.), devono prevedere l'introduzione di un ulteriore fattore casuale (salt) durante la loro generazione.

Batch Job dell'applicazione

Le informazioni, i dati o gli allegati trasmessi tramite i batch job dell'applicazione (ad esempio, sessioni ftp o altri protocolli di rete nativamente non cifrati o proprietari), devono utilizzare canali di comunicazione sicuri, come SSL o TLS, in cui le chiavi di cifratura simmetriche vengono scambiate all'interno di una comunicazione protetta attraverso algoritmo crittografico asimmetrico (ad esempio, RSA con dimensione delle chiavi uguale o superiore a 1024 bit).

Storage dei dati applicativi

I dati dell'applicazione memorizzati nel database o nel file system devono essere cifrati tramite algoritmi simmetrici con chiave pari almeno a 192 bit (inclusi i bit di parità).

Integrità delle informazioni

Tutti i dati di natura critica conservati e mantenuti dall'applicazione, oltre che cifrati, devono prevedere l'utilizzo di algoritmi di hashing o della firma digitale per poterne vagliare l'integrità/autenticità.

Meccanismi di autenticazione

L'applicazione sviluppata non deve impiegare meccanismi di autenticazione con chiave condivisa (altrimenti detti pre-shared secret).

Non ripudio delle sessioni

Tutte le sessioni riconducili all'applicazione svolte dagli utenti operativi e dall'amministratore devono essere, oltre che supportate da meccanismi di tracciamento idonei, anche cifrate con algoritmi crittografici atti a garantire il non ripudio delle singole sessioni. In altre parole deve essere possibile determinare con esattezza nel tempo l'occorrenza o la non occorrenza di un determinato evento.

Schemi di sicurezza e crittografici

Gli schemi di sicurezza devono essere semplici e ben documentati. È vietata la costruzione di schemi non-standard, e/o "hand-made", per l'autenticazione, la crittografia o e/o la gestione delle chiavi.

Weak Key e Collision

Il processo di creazione/assegnazione delle chiavi di cifratura dei dati dell'applicazione, in base al cifrario utilizzato, non deve generare weak key o, nel caso di algoritmi di hashing, collisioni.

URL cifrati

Le directory contenenti file o dati di natura personale, critica e sensibile, residenti nella Document Root del Web server, devono apparire come cifrate nell'URL del client browser.

Normalizzazione dei dati cifrati

Nelle applicazioni Web l'utilizzo della codifica base64 è autorizzata solo per la normalizzazione dei dati, delle stringhe o degli URL cifrati.

Tracciamento

Tracciamento eventi

L'applicazione deve essere predisposta al tracciamento delle attività svolte dall'utente e delle eccezioni che si sono verificate. Per ogni "evento" i log di audit devono indicare le seguenti informazioni:

- user id;
- data, ora e descrizione dell'evento/errore (ad esempio, log-on, log-off, etc.);
- identità di chi ha causato l'evento o l'errore (Indirizzo IP, FQDN).

La configurazione di cosa può essere tracciato è a discrezione dell'amministratore del sistema; tuttavia, la predisposizione al tracciamento degli eventi andati a buon fine, di quelli non andati a buon fine e degli errori deve, comunque, essere implementata nell'applicazione. Gli eventi per i quali è richiesta tale predisposizione riguardano:

1. l'autenticazione e processi correlati;
2. l'avvio e l'arresto delle componenti dell'applicazione;
3. le violazioni dei criteri o delle policy configurate;
4. le modifiche alle configurazioni dell'applicazione;

5. l'accesso (per l'inserimento, la modifica, la rimozione e la lettura) ai dati, ai file ed alle risorse dell'applicazione; in questo caso è necessario memorizzare anche la tipologia di accesso;
6. la disattivazione del meccanismo di tracciamento.

La procedura di tracciamento deve essere predisposta per l'emissione di alert al verificarsi di uno o più eventi configurabili dall'amministratore.

Utenze sottoposte a tracciamento

L'applicazione deve permettere il tracciamento dell'evento/errore occorso indipendentemente dal tipo di utente che lo ha scatenato (sia esso un operatore semplice o un amministratore).

2

La Code Inspection

In questo capitolo verrà illustrato il processo di ispezione del codice sorgente che rappresenta la nuova frontiera della sicurezza informatica. In particolare, ne verrà descritta la struttura, sottolineandone i benefici effetti. Particolare attenzione verrà rivolta a strumenti software che rendono automatica l'analisi del codice evidenziando i requisiti auspicabili. Il capitolo si chiuderà indicando una possibile soluzione al problema della certificazione del codice eseguibile.

2.1 Approccio al Problema della Sicurezza del Software

Autorevoli istituti di ricerca, tra i quali Gartner, indicano in una proporzione di circa tre a uno il numero di incidenti di sicurezza scaturiti da vulnerabilità di tipo applicativo. I problemi derivanti da questo tipo di vulnerabilità sono di gran lunga maggiori, sia in termini di numeri sia di efficacia, rispetto agli altri aspetti, seppure non trascurabili, che concorrono alla manifestazione di incidenti di sicurezza, siano essi legati a frodi, a tentativi di intrusione o a meri errori.

È, dunque, evidente che occorre sviluppare applicazioni sicure, ovvero rendendole robuste nei confronti di vulnerabilità caratteristiche globalmente riscontrate, già all'atto del loro impiego, vale a dire in fase di rilascio dopo il collaudo, oppure durante le fasi del loro sviluppo.

Per raggiungere tale scopo si potrebbero adottare varie metodologie come, ad esempio, il *Vulnerability Assessment*; con questo termine si intende il processo mediante il quale si identificano e quantificano le possibili debolezze nella sicurezza di un sistema o di un'applicazione, partendo dal suo "esterno" (blackboxing).

Un'altra tipologia di approccio al problema è la tecnica dello *Stress test*; essa è finalizzata alla verifica di applicazioni di tipo "Service Oriented", per saggiarne prestazione, robustezza e separazione logica.

Oggigiorno, però, la tendenza a concentrarsi sulla sicurezza del codice sorgente, ovvero sul cuore stesso dell'applicazione, si afferma sempre di più negli ambienti

degli esperti di sicurezza in quanto la sua analisi presenta il miglior rapporto costi/benefici e si prospetta come la nuova frontiera della sicurezza informatica.

In conformità alle nuove tendenze che si sono andate affermando negli anni recenti, e che hanno dimostrato di essere in grado di produrre risultati effettivi, ci si concentrerà, quindi, sul processo di analisi del codice sorgente che caratterizza le applicazioni, al fine di individuarvi eventuali debolezze nella programmazione, dal punto di vista sintattico e semantico, o direttamente sul flusso logico, in grado di essere sfruttate per modificare il comportamento dell'applicazione stessa.

Michael Fagan [4], nel 1976, fu il primo a definire formalmente il processo di ispezione del codice sorgente. Per questo motivo il processo di "Code Inspection" viene spesso denominato "Fagan Inspection". Più in generale, però, a lui si deve la definizione di processi strutturati per la ricerca di difetti nello sviluppo di documenti di vario tipo.

2.2 Scopo del Processo di Code Inspection

Un *processo* può essere descritto come un insieme di operazioni eseguite in una sequenza ben determinata che opera su un dato in ingresso e lo converte in un risultato desiderato. Una gestione efficace di qualsiasi processo richiede *pianificazione*, *misurazione* e *controllo*.

Nello sviluppo di applicativi software, tali requisiti si traducono nella definizione del processo di programmazione in termini di una serie di operazioni, ciascuna delle quali genera in uscita dei risultati. Contestualmente deve essere anche valutata l'integrità del prodotto in ogni momento del suo sviluppo attraverso ispezioni e test. Tale approccio non solo è concettualmente interessante ma è stato applicato con successo in tanti progetti di programmazione relativi a sistemi ed applicazioni, sia grandi che piccoli.

Per ridurre i costi legati alla rielaborazione degli errori nei programmi, ogni sforzo dovrebbe essere volto a trovare e correggere gli errori il prima possibile, durante lo sviluppo dell'applicativo, magari in prossimità del loro punto di origine.

L'ispezione del codice sorgente aumenta la produttività e la qualità del prodotto e costituisce un metodo formale, efficiente ed economicamente vantaggioso di individuare errori nel progetto e nel codice. Essa costituisce lo strumento più importante che si possa utilizzare per diminuire i tempi di rilascio del software e per ridurre considerevolmente il numero di vulnerabilità.

Il suo obiettivo è quello di identificare e rimuovere i difetti prima che il software venga testato. A titolo di esempio, IBM considera di rimuovere, tramite processi di ispezione sul codice, l'82% dei difetti complessivi prima che i test abbiano inizio. AT&T reputa che le ispezioni portino un incremento del 14% nella produttività ed un aumento pari a dieci volte della qualità del prodotto. HP stima che l'80% dei difetti individuati durante l'ispezione del codice non verrebbero scoperti durante le successive sessioni di test.

Un'ispezione può costare, in termini di tempo, circa il 20% del tempo dedicato alla programmazione, ma il tempo dedicato al debug può ridursi di un ordine di grandezza e anche di più.

2.3 Descrizione del Processo di Code Inspection

L'ispezione del codice sorgente che scaturisce dal modello di Fagan è un processo integrato al ciclo di sviluppo di un applicativo ed è poco, o per nulla, automatizzato. Esso implica l'impiego di una squadra di ispezione la cui efficacia è maggiore quando i membri che la compongono ricoprono ruoli ben precisi; Tali ruoli sono i seguenti:

- Il *moderatore* è la persona chiave in un'ispezione di successo. Egli deve essere un programmatore competente, ma non occorre che sia un tecnico esperto nel programma che sta per ispezionare. Per preservare l'oggettività e aumentare l'integrità dell'ispezione è generalmente vantaggioso che il moderatore venga prelevato da un progetto scorrelato.
  Il moderatore deve gestire la squadra e costituirne la guida; egli, quindi, deve utilizzare la propria sensibilità e deve essere dotato di tatto e senso della misura. Spetta a lui il compito di stimolare uno spirito di collaborazione e di sinergia. In altre parole, egli è il "coach".
  I doveri del moderatore includono un'opportuna pianificazione dei luoghi di incontro nonché la stesura di resoconti giornalieri dei risultati dell'ispezione e delle modifiche apportate al codice.
- Il *progettista* è il responsabile della realizzazione dell'architettura dell'applicazione.
- Il *programmatore* è il responsabile della traduzione del progetto in un codice sorgente che costituisce il software vero e proprio.
- il *verificatore* è il responsabile che effettua i controlli (test case) sul progetto e sul codice.

Una squadra costituita da quattro persone è ben dimensionata, anche se le circostanze potrebbero richiedere opportune modifiche. È importante evitare di ridurre il numero dei costituenti la squadra attraverso l'accorpamento dei ruoli; infatti, è stato dimostrato che un gruppo d'ispezione costituito da quattro persone è due volte più efficiente e valido di uno costituito da tre.

Poiché la capacità di rilevazione degli errori da parte della maggior parte dei gruppi di ispezione diminuisce dopo due ore di ispezione, ma migliora dopo un periodo di tempo in cui è stata svolta un'attività differente, è consigliabile pianificare sessioni di ispezione che non durino più di due ore per volta. Due sessioni di due ore ciascuna al giorno sono accettabili.

Il tempo dedicato alle ispezioni ed alla conseguente rielaborazione del codice deve essere pianificato e gestito con la stessa attenzione riservata alle altre importanti attività di progetto. Spesso la contingente pressione del lavoro tende a mettere in secondo piano le ispezioni e le modifiche, rimandandole o, addirittura, evitandole del tutto.

L'effetto di un eventuale rinvio sarà, ovviamente, molto negativo, non solo nel periodo che intercorre tra la scoperta e la correzione degli errori; esso potrebbe comportare, in ultima analisi, un aumento dei costi ed un allungamento dei tempi di rilascio.

L'ispezione del codice sorgente è descritta da una sequenza di sei passi, tutti necessari per l'ottenimento di risultati ottimali; tali passi, riportati in Figura 2.1, verranno illustrati nelle prossime sottosezioni.

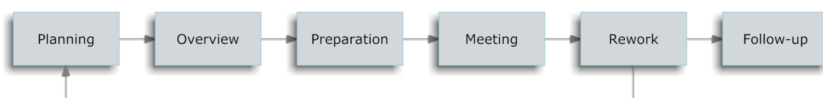


Figura 2.1. Diagramma di flusso di una Fagan Inspection

2.3.1 Planning

In questa fase il moderatore prepara il materiale informativo, forma una squadra di ispezione e stabilisce i luoghi di incontro.

2.3.2 Overview

Questo passo coinvolge l'intera squadra ed ha l'obiettivo di istruirla. La documentazione redatta dal progettista è distribuita a tutti i partecipanti al processo di ispezione. Essa descrive inizialmente la struttura del progetto e, successivamente, scende nei dettagli evidenziando le logiche di funzionamento, i percorsi, le librerie, etc.

2.3.3 Preparation

Anche questo passo ha l'obiettivo di istruire la squadra, ma, in questo caso, viene svolto individualmente. I partecipanti, tramite la documentazione loro fornita, svolgono indipendentemente il compito di capire la struttura, la logica e gli obiettivi del software.

Spesso accade che in questa fase vengano riscontrati errori grossolani anche se, in generale, il numero di errori non è minimamente paragonabile a quello ottenuto

dopo un'operazione di ispezione vera e propria. Per aumentare il numero di errori rilevati durante l'ispezione, il team dovrebbe dapprima studiare le tabelle che indicano le distribuzioni dei tipi di errore individuati nelle precedenti sessioni di ispezione. Tale studio suggerirà loro di concentrare la propria attenzione su aree particolarmente esposte.

2.3.4 Inspection Meeting

Questa fase, che costituisce il cuore del processo, coinvolge l'intera squadra ed ha l'obiettivo di individuare gli errori nel codice.

Un "lettore" scelto dal moderatore (generalmente il programmatore) descrive l'implementazione del progetto parafrasando il significato di piccole sezioni di codice in un contesto di più alto livello rispetto al codice stesso.

Tutta la documentazione relativa alle specifiche di progetto, funzionamento logico, etc. deve essere disponibile durante l'ispezione.

Le domande sollevate durante l'esposizione del lettore permettono di pervenire alla scoperta di errori che vengono annotati e classificati dal moderatore in base alla loro minore o maggiore gravità. Durante questa fase il rigore è garantito tramite l'uso di checklist.

È importante sottolineare che l'ispezione non intende valutare soluzioni alternative al progetto o indicare soluzioni agli errori; essa deve essere tesa esclusivamente alla scoperta degli errori. D'altro canto, una squadra è efficiente se lavora perseguendo un singolo obiettivo per volta.

A incontro concluso, il moderatore redigerà un resoconto dei risultati dell'ispezione che servirà per le fasi successive.

2.3.5 Rework

Coinvolge il progettista e/o il programmatore ed ha l'obiettivo di correggere tutti gli errori evidenziati nel report dell'ispezione.

2.3.6 Follow-up

Ha l'obiettivo di assicurare che tutte le modifiche siano state apportate correttamente. È compito del moderatore assicurarsi che, durante questa fase, ogni errore venga categoricamente risolto; altrimenti, effettuando le correzioni successivamente, si spenderà del tempo da 10 a 100 volte maggiore.

Se è stato rielaborato più del 5% del codice, la squadra dovrà eseguire una nuova ispezione del progetto nella sua completezza. Se è stato modificato meno del 5% del codice, il moderatore può verificare, a sua discrezione, la qualità delle modifiche apportate al progetto oppure disporre una nuova ispezione del progetto nella sua interezza, o soltanto delle modifiche apportate.

Uno dei benefici più significativi delle ispezioni è la tempestività del riscontro degli errori. Il programmatore, così, si accorge del tipo di errori che è propenso a commettere, della loro quantità e del metodo più opportuno per rilevarli. Questo risultato si ottiene dopo pochi giorni di scrittura del codice. Grazie a tali informazioni iniziali, ottenute dopo poche unità di ispezione del suo lavoro, il programmatore potrà conseguire miglioramenti non solo per i progetti futuri, ma anche per quello in corso.

2.4 Strumenti Software

Oggigiorno capita spesso che il processo di ispezione del codice sia “esterno” al processo di sviluppo dello stesso e che la squadra che esegue l’ispezione sia più numerosa ed articolata.

Dati i volumi in gioco, la sensibilità dei dati trattati, il numero di fornitori e, soprattutto, i tempi ridottissimi di entrata in produzione delle applicazioni, emerge chiaramente il bisogno di ricorrere all’ausilio di strumenti di analisi automatici o semi-automatici. Oggi, quando si parla di Code Inspection (whiteboxing) si intende indicare la fase di analisi del codice sorgente finalizzata ad eliminare, sulla base di un database di regole note e storicamente documentate, eventuali debolezze dovute all’uso di particolari costrutti semantici, di funzioni, oggetti, strutture dati o metodi del linguaggio in esame che possano essere sfruttate per sovvertire un’applicazione.

Sulla base dei principi di sicurezza per il codice sorgente descritti nel capitolo precedente si evince la necessità che il software scelto per effettuare l’analisi del codice risponda alle seguenti caratteristiche:

- controllo sulla validazione dell’input;
- controlli su bound checking ed overflow;
- controlli sulle sessioni;
- controlli sulla crittografia e sulla robustezza delle password;
- questione dei problemi di timing e delle debolezze risultanti.

In Figura 2.2 è riportato un semplice modello di analisi del codice sorgente.

Alcuni software di carattere commerciale sono di seguito illustrati.

Nelle prossime sottosezioni presenteremo alcuni software commerciali pensati per effettuare tale controllo.

2.4.1 Coverity Prevent

Coverity Prevent è un prodotto di analisi in senso assoluto. L’interfaccia è molto semplice, il suo utilizzo richiede la scrittura di regole che permettono la diminuzione di falsi positivi; esso consente anche l’inserimento, durante la stesura del

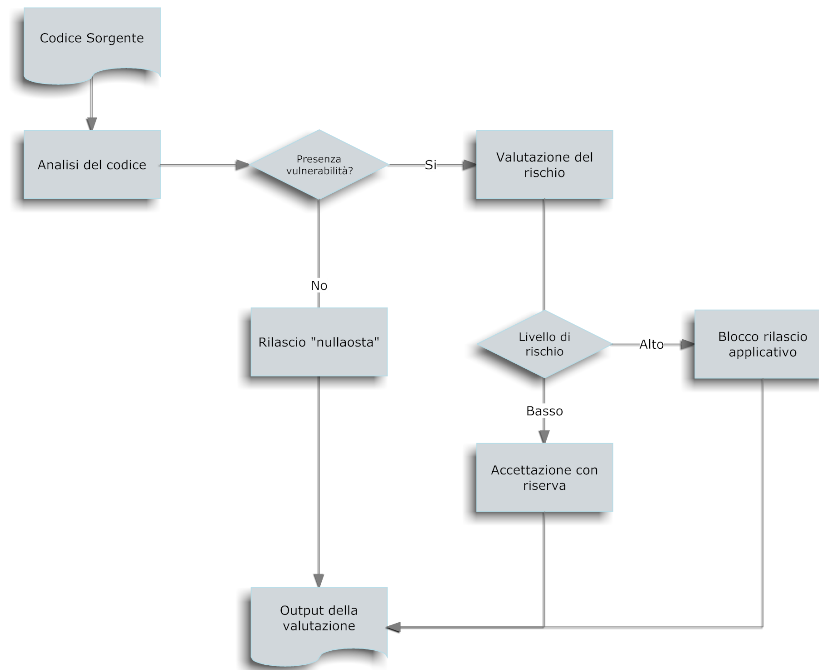


Figura 2.2. Diagramma di flusso di un semplice modello di analisi del codice sorgente

codice sorgente, di commenti che permettono di migliorare l'analisi nelle fasi di certificazione.

Tale software viene distribuito con plug-in che si adattano ad alcuni degli ambienti di sviluppo (IDE) più diffusi (ad esempio, Eclipse) ed è dotato di un "Defect Manager" che permette di notificare agli utenti, mediante mail o tramite uno strumento di bug-tracking, i bug individuati. Coverity Prevent può essere utilizzato su piattaforme: Windows, Linux, Solaris (SPARC and x86), HP-UX, NetBSD, FreeBSD, ma supporta solo due linguaggi: C e C++. Infine, il prodotto ha una licenza d'uso commerciale con limitazioni sul numero di righe di codice ispezionabili.

2.4.2 Fortify Source Code Analysis Suite

Fortify Source Code Analysis Suite è una suite multi-componente che comprende un motore d'analisi, vari plug-in per l'integrazione in ambienti di sviluppo (IDE), un editor delle regole di analisi e, opzionalmente, un portale Web per i riepiloghi delle analisi.

Il motore della suite software è in grado di processare i codici sorgente dei linguaggi C, C++, C#, Java, JSP, PL/SQL (Oracle) e TSQL (Microsoft). Esso opera in modalità consolle (da riga di comando) e, pertanto, risulta particolarmente adatto all'impiego in modalità "remota" o tramite scripting.

Una volta effettuata l'analisi sarà possibile generare resoconti particolarmente dettagliati o, laddove necessario, accedere alla base di dati per l'estrapolazione di informazioni specifiche.

Da evidenziare è la completezza dei report generati, con particolare riferimento alle informazioni mostrate per ciascuna possibile vulnerabilità individuata che comprendono, tra le altre cose, la spiegazione dei rischi e le possibili soluzioni. A tale proposito, vengono utilizzati frammenti di codice prelevati dall'applicazione sotto indagine.

Il prodotto ha una rosa di tipologie di licenze commerciali non sottoposte a limiti nel numero di righe di codice.

È interessante notare che la suite Fortify dispone anche di due prodotti complementari all'ispezione del codice sorgente.

Il primo è uno strumento software di vulnerability assessment che sfrutta i risultati ottenuti dall'analisi del codice di un'applicazione Web per sondare il front-end ricavandone l'entità dei danni che si possono ottenere (grayboxing). In questo modo si rivoluzionano le tecniche di penetration test/vulnerability scanning operando "sul sicuro", nell'ambito di vulnerabilità identificate. Un secondo strumento, anch'esso disponibile solo per le applicazioni Web, ha lo scopo di proteggere automaticamente tutte le vulnerabilità rilevate.

2.4.3 Secure Software CodeAssure

Secure Software CodeAssure è un ambiente di analisi per C, C++ e Java. Esso viene fornito come un plug-in da integrare ad una installazione di Eclipse.

L'analisi delle applicazioni viene effettuata su una sorta di "macchina virtuale" e, quindi, è richiesto che i file sorgente siano compilabili ed eseguibili.

Questo prodotto software offre il più vasto repository di documentazione sui bug individuati ma, allo stesso tempo, pecca alle volte di genericità sui rimedi proposti.

Anche in questo caso il prodotto è dotato di licenza commerciale che prescinde dal numero di righe di codice ispezionabili, ma che presenta la necessità di iscrizioni annue.

Infine, va sottolineato che Secure Software è stata recentemente acquisita da Fortify Software.

2.4.4 Klocwork K7

L'ambiente integrato di K7 offre numerosi strumenti indipendenti; si va dal *Klocwork Management Center*, che permette di effettuare un browsing "centralizza-

to" della creazione dei progetti di analisi e della loro configurazione, al *Project Central Web Portal*, che permette di avere riepiloghi dettagliati nonché analisi architetturali mirate a supportare la comprensione e la gestione del codice.

Per il suo funzionamento si richiede che vengano realizzate delle "library extensions" che verranno compilate come una qualsiasi libreria condivisa (o meno) dell'applicazione.

In sintesi, Klocwork K7 è una suite completa di analisi corredata da vari strumenti di controllo. Esso opera su C, C++ e Java (sorgente e bytecode), ma è limitato a tre sole tipologie di piattaforme: Windows, Linux, Solaris.

Il prodotto è venduto con licenza commerciale che ne limita fortemente l'uso. Dal punto di vista dell'usabilità, la creazione dei progetti e la loro successiva gestione risulta essere macchinosa e poco adatta alla gestione di un elevato numero di applicazioni. Inoltre, K7 presenta un notevole svantaggio per quanto riguarda la raccolta e l'analisi delle possibili vulnerabilità; infatti, poiché è necessario definire tipologia in fase di creazione del progetto le tipologie di vulnerabilità da processare, una modifica di tali impostazioni comporterebbe la necessità di effettuare una nuova scansione, con la conseguente moltiplicazione dei tempi necessari all'ottenimento dei risultati.

2.4.5 Ounce Labs Prexis

Prexis è simile a K7 della Klocwork e ai prodotti della Fortify in quanto il motore di analisi può essere lanciato direttamente da riga di comando.

Ounce Labs non offre plug-in per interfacciare il prodotto ad ambienti di sviluppo, ma, per la visualizzazione del codice sorgente, possono essere configurati ed utilizzati editor di propria scelta.

Il motore di analisi può essere configurato con delle "customized rules"; per avere strumenti di rendicontazione centralizzati occorre acquistare una "department installation" con licenze per cinquanta sviluppatori e due manager.

Per quanto riguarda l'interfaccia, seppur completa dal punto di vista delle informazioni presentate, emerge una non semplice leggibilità delle stesse a causa dei numerosi pannelli e riquadri visualizzati.

L'usabilità del prodotto risente pesantemente di tale complessità. Particolarmente rilevante, invece, è la mancanza di procedure che consentano di modificare la severità delle diverse vulnerabilità individuate che, di fatto, potrebbe considerevolmente limitare l'ottimizzazione delle procedure di analisi successive. L'unico sistema per ovviare a tale problematica consiste nella creazione di nuove regole-clone, personalizzate per ciascuna regola già esistente. Tale compito risulta essere estremamente oneroso e fonte di eventuali nuovi errori.

Altrettanto limitante è la knowledge base presente che, seppur contenendo un buon numero di informazioni ed esempi, non fornisce indicazioni e riferimenti a fonti esterne; quest'ultima peculiarità sarebbe estremamente utile al fine di garantire un'analisi migliore e più corretta delle possibili vulnerabilità individuate.

2.5 Requisiti e vincoli

Nel seguito presenteremo le caratteristiche che dovrebbe possedere un software ideale di code inspection. A seconda del contesto applicativo, alcune di queste dovranno risultare obbligatoriamente soddisfatte, dal momento che caratterizzano ciò che il prodotto deve fare; altre, invece, saranno facoltative perché associate a funzioni minori, che sarebbe comunque preferibile avere, e, soprattutto, alle modalità con cui tali funzioni devono essere effettuate.

2.5.1 Requisiti tecnologici

1. *Possibilità di definire regole personalizzate*: deve essere possibile la definizione di regole di ispezione personalizzate, in modo da conformarsi alle esigenze immediate e a quelle future.
2. *Possibilità di selezionare le regole da verificare*: assumendo la presenza di un elevato numero di regole di verifica generali, deve essere possibile selezionare da queste un sottoinsieme da utilizzare per l'ispezione.
3. *Possibilità di alterare il livello di classificazione delle regole*: a seconda delle strategie di protezione che si vogliono adottare e che si andranno via via affinando, deve essere possibile classificare le regole in maniera personalizzata.
4. *Capacità di ispezione sul maggior numero possibile di linguaggi*: il sistema deve essere aperto e riuscire a coprire più linguaggi.
5. *Funzioni di autenticazione e tracciamento*: a tutela della proprietà intellettuale e della sensibilità delle informazioni trattate, l'accesso al sistema da parte degli operatori deve richiedere una forma di autenticazione insieme ad un tracciamento delle operazioni svolte nonché delle impostazioni e dei risultati ottenuti.
6. *Aggiornamenti delle regole*: il produttore deve fornire aggiornamenti sia modificando eventualmente le regole di verifica esistenti sia aggiungendone delle nuove in base ai risultati della ricerca.
7. *Ambienti di esecuzione standard*: i client devono poter girare su piattaforme Microsoft Windows in architettura x86, mentre gli eventuali server dovranno operare sulle piattaforme e le architetture più diffuse in ambito aziendale (Intel, Sparc, Power5).
8. *Accesso internet centralizzato*: gli eventuali aggiornamenti dovrebbero essere scaricati centralmente, da parte di un server.

2.5.2 Requisiti organizzativi

1. *Massimo automatismo*: il sistema, una volta impostato, deve richiedere la minore interazione possibile con l'operatore.
2. *Semplicità di operazione*: il sistema, una volta impostato e configurato, non deve richiedere ampie competenze da parte dell'operatore.

3. *Riepilogo dettagliato*: il sistema deve essere in grado di generare un riepilogo quanto più dettagliato possibile del risultato dell'ispezione.
4. *Riepilogo illustrativo*: il riepilogo che il sistema genera come risultato dell'ispezione deve illustrare ogni vulnerabilità riscontrata, spiegandone i rischi o facendo riferimento a codici o "best practice" internazionalmente riconosciute.
5. *Riepilogo personalizzabile*: il riepilogo che il sistema genera come risultato dell'ispezione deve essere personalizzabile per quanto riguarda lo stile e i contenuti.
6. *Architettura client/server*: il sistema deve essere realizzato secondo un'architettura client/server, in modo da centralizzare il caricamento dei dati in ingresso, delle impostazioni e delle configurazioni.
7. *Architettura multiutente*: il sistema deve essere realizzato secondo un'architettura multi-utente con accessi contemporanei.
8. *Contenitore del codice e delle versioni*: il sistema deve essere in grado di contenere il codice analizzato, gestendone le diverse versioni e i risultati delle ispezioni.
9. *Forte controllo d'integrità*: sul sistema deve essere presente a tutti i livelli un meccanismo di controllo dell'integrità sia del codice trattato sia delle regole di verifica, dei risultati e delle impostazioni.
10. *Firma su date e operatori*: deve essere possibile garantire le operazioni svolte sul sistema in modo non opponibile dalla legge.
11. *Profili centralizzati*: sul sistema deve essere possibile impostare centralmente i profili e i diritti di ogni operatore relativamente alle operazioni, al codice e ai risultati.
12. *Impossibilità di esportare il codice*: una volta caricati i dati in ingresso, il sistema non deve consentire a un qualunque operatore di esportare il codice sorgente nella sua interezza.

2.6 La certificazione del codice eseguibile

Il bisogno di verificare la corrispondenza tra il codice sorgente e il codice eseguibile va assumendo attualmente una particolare importanza soprattutto quando il software viene realizzato in outsourcing da linee di sviluppo esterne.

Sfortunatamente non è possibile firmare il sorgente e verificare la firma sull'eseguibile in quanto le complesse trasformazioni dai linguaggi ad alto livello al linguaggio macchina operate dai compilatori trasformano completamente il listato eliminando le parti non necessarie all'esecuzione delle istruzioni. Per tale ragione non si riuscirebbe a ritrovare una firma nel codice eseguibile.

D'altra parte non è nemmeno possibile confrontare l'hash di due eseguibili ottenuti dallo stesso codice sorgente, dal momento che questi differiscono in ogni caso perché ogni compilatore inserisce delle informazioni variabili relative,

ad esempio, alla data e all'ora di compilazione, al suo numero di serie, alla licenza d'uso, e così via.

Nel seguito vediamo quali sono le possibili soluzioni a tali problematiche.

2.6.1 Compilazione in ambiente interno/controllato

Il primo scenario, il più immediato, consiste nell'allestire internamente un gruppo dedicato alla compilazione di tutte le applicazioni prodotte all'esterno. Ogni fornitore dovrebbe consegnare la documentazione, il materiale e tutto quanto necessario alla compilazione.

Dal punto di vista della sicurezza, questa ipotesi è da giudicare ottima in quanto consente di essere assolutamente certi della corrispondenza tra sorgente ed eseguibile, a meno di librerie già compilate da associare.

Chiaramente tale soluzione comporta, però, gravi impatti sul processo e sui relativi costi. Un tale gruppo dovrebbe essere numeroso, composto da analisti programmatori specializzati in tutti i linguaggi effettivamente impiegati; si dovrebbe disporre di una moltitudine di compilatori e piattaforme, con le corrispettive licenze, con le corrispondenti librerie, i corrispettivi oggetti e altre componenti costose; tutte queste cose, inoltre, sarebbero tipicamente legate al mondo delle software house con tanto di diritti di esclusiva e altre limitazioni.

Inoltre, si incontrerebbero problemi sui parametri di compilazione e sorgerebbero contenziosi tra la struttura interna, che non sarebbe in grado di procedere, e i fornitori. Infine, i tempi verrebbero necessariamente allungati, pregiudicando la messa in servizio degli applicativi.

Per tali ragioni lo scenario è difficilmente realizzabile, estremamente costoso e poco compatibile con i brevi tempi di rilascio richiesti.

2.6.2 Ispezione del decompilato

Il secondo scenario sposta l'attenzione direttamente sull'eseguibile, analizzando non il codice sorgente bensì il codice estratto dall'eseguibile, fornito tramite procedimento di disassemblaggio e decompilazione.

Tale soluzione porterebbe agli stessi risultati dello scenario precedente, senza tuttavia comportarne gli enormi svantaggi.

Sfortunatamente diversi aspetti la rendono inapplicabile. Difatti, il procedimento di decompilazione non è una trasformazione biunivoca, ovvero non rende lo stesso sorgente inizialmente fornito. I decompilatori sono pochi, approssimativi ed estremamente complessi.

Se prendiamo, come esempio, un lavoro estremamente più semplice, quale quello del disassemblaggio, vediamo come, con strumenti dedicati, per riuscire a disassemblare un'applicazione semplice siano necessarie diverse settimane e competenze molto specifiche, il tutto con una mediocre probabilità di successo.

Per questi motivi, l'ispezione del codice oggetto, nella migliore delle ipotesi, servirebbe a validare il decompilatore stesso, ma non l'eseguibile fornito in ingresso.

2.6.3 Ispezione a campione

Il terzo scenario si basa sull'idea di fornire una garanzia a campione. In questo caso, seguendo un piano di ispezioni, ed in accordo con quanto stipulato contrattualmente, si procede presso il fornitore ad una nuova compilazione del codice sorgente negli stessi ambienti che hanno prodotto l'eseguibile già consegnato, per poi svolgere una comparazione tra questo e l'eseguibile appena ottenuto.

Fermo restando che, in alcuni casi, non si riuscirebbe ad ottenere comunque un eseguibile identico (si pensi, ad esempio, a compilatori che inseriscono un numero progressivo oppure la data e l'ora di compilazione), l'ipotesi è praticabile, ma con tutti i limiti derivanti dal lavoro a campione, ovvero la mancata copertura completa.

Inoltre, i costi potrebbero diventare rilevanti nel caso in cui i fornitori operino in sedi situate presso altre città oppure presso altre nazioni, rendendo la soluzione difficilmente attuabile in concreto.

2.6.4 Cenni sulla ricerca delle invarianti

Un possibile approccio futuro, dato che il tema è ancora oggetto di studio e ricerca, si basa su tecniche di marcatura del codice sorgente e di analisi del codice eseguibile.

Alcune strutture di programmazione di livello più basso sono facilmente identificabili e, in talune circostanze, la loro distanza relativa può rimanere costante.

Si procede inserendo meta-dati in posizioni immediatamente successive ai puntatori. Programmando in modo da concatenare tra di loro i puntatori, è mediamente possibile ritrovare nell'eseguibile una meta-struttura che si paragona, poi, a quella di marcatura.

Di fatto, però, la tecnica presenta vari inconvenienti; infatti, attualmente non si dispone di strumenti diversi da quelli di proof-of-concept, si è strettamente legati al linguaggio di programmazione utilizzato e al tipo di codice scritto e, soprattutto, si è soggetti ad errore in quanto ci si basa su un modello statistico per cui non vi potrà mai essere la certezza della corrispondenza tra sorgente ed eseguibile. Siccome, però, la tecnica tenta di rispondere agli stessi requisiti formali di nostro interesse, sarà utile seguire la sua evoluzione nel medio e lungo termine.

2.6.5 Comparazione a campione negli ambienti di origine

Lo scenario più promettente consiste nel farsi consegnare da ogni fornitore, oltre alla documentazione, al codice sorgente e al codice eseguibile, anche un file contenente l'intero ambiente virtuale all'interno del quale tale codice eseguibile è stato realizzato.

Di fatto, si tratta di utilizzare gli appositi strumenti di VMWare che consentono di generare un file eseguibile come macchina virtuale partendo da un file system Microsoft Windows. In questo modo si disporrebbe del binomio sorgente-eseguibile insieme all'esatta e completa replica dell'ambiente di compilazione potendo, di conseguenza, svolgere le opportune verifiche di conformità. Tali verifiche potrebbero essere svolte a campione, ciclicamente oppure sistematicamente su tutte le applicazioni di nuovo rilascio.

È da notare, però, che i sistemi di virtualizzazione VMWare riescono ad emulare esclusivamente ambienti in architettura CISC Intel x86 o equivalenti, per cui compilazioni eventualmente svolte su altre architetture RISC (SPARC, POWER5, PA-RISC) non potrebbero essere verificate.

2.6.6 Considerazioni conclusive

Da quanto esposto non emerge una chiara soluzione, direttamente applicabile a tutti i casi e in grado di comportare una copertura completa. Tramite una combinazione di verifiche sugli ambienti virtuali e, quando ciò non è possibile, tramite ispezioni a campione svolte presso i fornitori, si potrebbe riuscire a coprire la maggior parte dei casi, affrontando poi le situazioni particolari che non vi possono rientrare.

Ci si potrebbe, comunque, concentrare inizialmente soltanto sulle applicazioni più critiche dal punto di vista del business e più sensibili da quello della sicurezza, riducendo, così, drasticamente il parco di applicativi da verificare; in questo modo si riuscirebbero ad attuare verifiche complete sia del codice sorgente sia della sua corrispondenza con l'eseguibile prodotto.

3

Il software Fortify

Il presente capitolo ha lo scopo di illustrare il software Fortify Source Code Analysis Suite evidenziandone soprattutto le caratteristiche che lo hanno reso l'applicativo di riferimento nell'ambito della sicurezza del codice sorgente. La descrizione di questo software verrà effettuata illustrando in dettaglio i singoli componenti.

3.1 Introduzione

Tutti i software descritti nel capitolo precedente, seppure in modo sensibilmente differente, risentono della tipica procedura utilizzata per l'ispezione del codice sorgente che, nella stragrande maggioranza dei casi, è parte integrante del ciclo di sviluppo. Tale impostazione logica traspare, inoltre, anche per quanto concerne la gestione di un elevato numero di progetti, non sempre agevole e immediata.

Nel nostro contesto applicativo, un elevato livello di flessibilità è un requisito fondamentale per la scelta del prodotto più adatto, anche in funzione dell'elevato numero di applicativi che saranno sottoposti a verifica ed analisi.

Altri requisiti di cui tener conto sono l'usabilità generale del prodotto, il livello di personalizzazione dei processi di analisi e l'esportazione ed intelligibilità dei risultati ottenuti. Particolare importanza riveste anche la verifica delle possibilità di automazione dei processi di analisi, della scansione del codice e della successiva raccolta centralizzata dei risultati.

Tra i prodotti illustrati nel capitolo precedente Fortify Source Code Analysis Suite (nel seguito, Fortify SCAS) è risultato essere quello caratterizzato da un maggiore livello di adattabilità ed astrazione dal processo di sviluppo, riuscendo ad individuare un cospicuo numero di vulnerabilità anche in presenza di singole porzioni di codice, in assenza di tutto il codice a contorno e nell'impossibilità di compilare correttamente l'applicazione. Occorre, comunque, evidenziare che lo scenario appena descritto risulta, tuttavia, particolarmente limitativo e non ottimale per una corretta ed esaustiva analisi ed individuazione di possibili vulnerabilità.

Fortify SCAS mostra anche la migliore usabilità per quanto riguarda l'impiego di strumenti a linea di comando, sia in ambiente Microsoft sia in ambienti Unix-like; tali strumenti sono particolarmente idonei per la realizzazione di script di controllo delle procedure consentendo il raggiungimento di un elevato livello di automazione delle procedure di analisi.

Una volta effettuata l'analisi sarà possibile generare resoconti particolarmente dettagliati oppure estrapolare informazioni specifiche tramite la base di dati contenuta nel software stesso. Da evidenziare è la completezza dei report generati, con particolare riferimento alle informazioni mostrate per ciascuna possibile vulnerabilità individuata; tali informazioni comprendono, tra le altre cose, la spiegazione dei rischi e le possibili soluzioni. A tale proposito vengono utilizzati frammenti di codice prelevati dall'applicazione sotto indagine.

La struttura modulare di Fortify SCAS permette una maggiore integrazione con componenti software di terze parti e facilita la scrittura di regole personalizzate.

Il prodotto vanta una rosa di tipologie di licenze commerciali non sottoposte a limiti nel numero di righe di codice, il che lo rende ancora più appetibile.

In sintesi, possiamo affermare che Fortify SCAS è il software che meglio soddisfa i requisiti ed i vincoli riportati nella Sezione 2.5.

A conferma di quanto scritto, già dal 2005, Oracle, dopo anni di utilizzo di un proprio software per eseguire attività di code inspection, ha deciso di optare per Fortify SCAS nella ricerca di possibili vulnerabilità e problemi di sicurezza del proprio software.

Secondo quanto dichiarato da Mary Ann Davidson, Chief Security Officer di Oracle, si tratta di una scelta quasi obbligata, poiché Fortify SCAS sembra essere il solo capace di analizzare il codice sorgente, di ingenti dimensioni, degli applicativi Oracle.

Oracle si affianca così a Macromedia e Oblix (acquisita in marzo 2005 da Oracle stessa) nel parco dei clienti di Fortify, popolato soprattutto da grosse società di servizi finanziari.

3.2 Fortify Source Code Analysis Suite

La struttura di Fortify SCAS è riportata in Figura 3.1.

Tale software può essere integrato nel ciclo di sviluppo del software di organizzazioni di varia tipologia e dimensione; esso può anche essere utilizzato da uno o più gruppi centralizzati di sicurezza del software.

Per venire incontro a tali disparate esigenze, esistono tre edizioni di Fortify SCAS; esse sono: l'Enterprise Edition (EE), la Developer Edition (DE) e la Team Edition (TE).

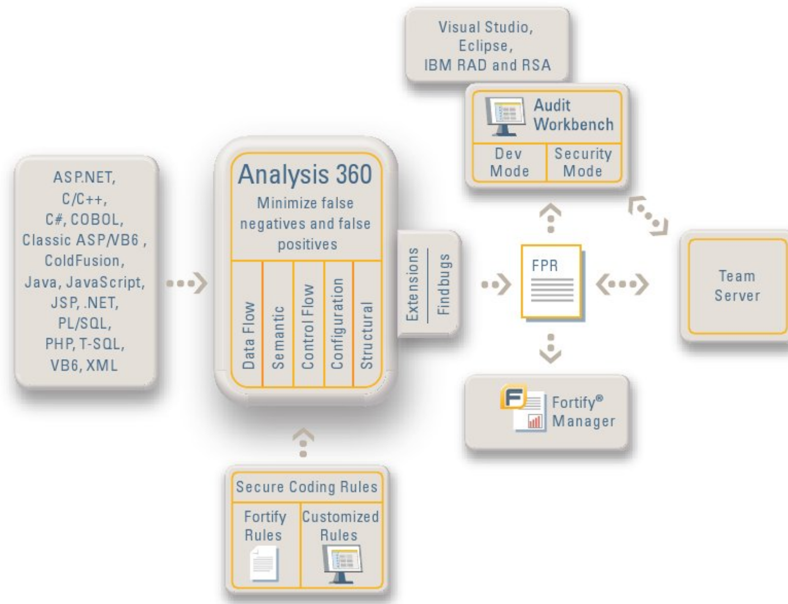


Figura 3.1. Struttura del Fortify Source Code Analysis Suite

L'*Enterprise Edition* rappresenta l'edizione più completa essendo costituita da ben cinque componenti; essi sono: il Fortify Source Code Analyzer, l'Audit Workbench, il Fortify Manager, il Secure Coding Rulepacks ed il Rules Builder.

L'edizione *Developer Edition* consta del Fortify Source Code Analyzer e del Secure Coding Rulepacks e viene corredata dal Secure Coding Plug-ins; quest'ultimo consiste in un insieme di plugin per software di sviluppo quali Eclipse, IBM Application Developer e Microsoft Visual Studio.

L'edizione *Team Edition* è raccomandata alle piccole organizzazioni essendo costituita solo dal Fortify Source Code Analyzer, dall'Audit Workbench e dal Rules Builder.

L'edizione da noi utilizzata nell'ambito della presente tesi è l'Enterprise Edition versione 4.5. Occorre, comunque, evidenziare la continua evoluzione del software che si arricchisce di nuove caratteristiche, come il crescente numero di linguaggi supportati, gli aggiornati strumenti di aiuto per la scrittura di regole personalizzate e le innovative modalità di impiego.

Proprio recentemente Fortify Software ha annunciato la sua ultima Suite di sicurezza, denominata "Fortify 360"; questa evidenzia già dal nome l'approccio a 360 gradi al problema della sicurezza del software.

3.2.1 Fortify Source Code Analyzer

Fortify Source Code Analyzer (nel seguito, Fortify SCA) rappresenta il cuore della suite Fortify, essendo il componente utilizzato per l'acquisizione e l'analisi del codice sorgente.

Il suo funzionamento è basato su un insieme di analizzatori specializzati nell'individuazione di specifiche violazioni di regole e di linee guida di sviluppo del codice.

Si distinguono i seguenti cinque specifici analizzatori:

- *Data flow*: consente la rilevazione delle criticità legate principalmente al flusso dei dati, con particolare attenzione ai punti di ingresso che coinvolgono gli utenti.
- *Control flow*: consente la rilevazione dei concatenamenti di funzioni e processi interni all'applicativo che potrebbero portare al verificarsi di problematiche di sicurezza.
- *Semantic*: consente la rilevazione di utilizzi potenzialmente critici di funzioni e/o API a livello intra-procedurale.
- *Structural*: consente la rilevazione delle criticità presenti in un programma a livello strutturale.
- *Configuration*: consente la rilevazione di errori, debolezze e violazioni di policy all'interno di file di configurazione.

Ciascuno di questi analizzatori gestisce una diversa tipologia di firme di rilevamento, in relazione alla tipologia di "osservazione" effettuata. In tal modo sarà possibile focalizzare l'analisi su specifiche tipologie di problematiche.

All'individuazione delle vulnerabilità succede la loro classificazione in modo tale che le correzioni apportate al codice sorgente siano più rapide e accurate.

Il Fortify SCA permette, così, un rilascio di software più sicuro, ma anche revisioni del codice più efficienti, consistenti e complete, specialmente quando il codice sorgente è di notevoli dimensioni.

Fortify SCA è in grado di processare i codici sorgente dei linguaggi C, C++, C#, Java, JSP, PL/SQL (Oracle) e TSQL (Microsoft); nelle ultime versioni del software è stato aggiunto il supporto ai linguaggi ASP.NET, COBOL, Classic ASP/VB6, ColdFusion, PHP ed XML. Fortify SCA opera in modalità consolle (da riga di comando) e, pertanto, risulta particolarmente adatto all'impiego in modalità "remota" o tramite scripting.

La sua versatilità si manifesta anche nella varietà di piattaforme su cui può essere eseguito; esse sono Windows, Solaris, Linux, Mac OS X, HP-UX e AIX.

Il processo di analisi tramite Fortify SCA consiste delle seguenti tre fasi:

1. *Traduzione*: il codice sorgente, dopo che ad esso è stato associato un Build-ID, deve essere tradotto in un formato intermedio. Il formato intermedio utilizzato dallo strumento è identificato dall'estensione `.nst`.

Usualmente il Build-ID è il nome del progetto da sottoporre a scansione ed è normalmente utilizzato per identificare una specifica sessione di analisi. Il suo impiego è fondamentale quando è necessario suddividere l'attività di acquisizione in diversi passaggi. Infatti, nel caso in cui l'applicazione fosse realizzata da diversi moduli, sviluppati utilizzando diversi linguaggi di programmazione, sarà necessario procedere con la traduzione separata di ciascun modulo utilizzando in tutti i passaggi il medesimo Build-ID.
La sintassi di base, da utilizzare attraverso linea di comando è:

```
sourceanalyzer -b <build-id> ...
```

Si può aggiungere alla sintassi precedente il parametro opzionale `-show-build -warnings`, per mostrare nello schermo eventuali segnalazioni generate durante il processo di traduzione, oppure il parametro `-show-files`, per visualizzare tutti i file associati ad uno specifico Build-ID.

2. *Scansione*: i file sorgente tradotti durante la fase precedente vengono sottoposti a scansione; in questo modo viene generato un file contenente tutti i risultati di questo processo (il formato tipicamente utilizzato è “Fortify Project”, identificato dall'estensione `.fpr`).
La scansione consiste in una o più invocazioni di `sourceanalyzer` in cui viene indicato il Build-ID e viene specificata l'opzione `-scan`.
Un esempio di sintassi utilizzata per una scansione è il seguente:

```
sourceanalyzer -b <build-id> -scan -f result.fpr -logfile file.log
```

Si noti inclusione dell'opzione `-logfile` che permette la creazione di un file di log all'interno della stessa directory in cui si sta operando.

3. *Verifica*: viene accertato che i file sorgente siano stati sottoposti ad analisi utilizzando le regole corrette e che non siano stati riportati errori significativi. Il file di log creato utilizzando il comando precedente contiene tutte le informazioni relative all'esecuzione del processo. Prima di proseguire con le successive fasi di analisi è necessario verificare che al suo interno non siano riportati errori o segnalazioni (ad esempio, l'impossibilità ad accedere a specifici file, l'assenza di librerie, etc.). Tutte le problematiche riscontrate dovranno essere affrontate o risolte prima di procedere con i passi successivi.

3.2.2 Fortify Audit Workbench

Fortify Audit Workbench affianca Fortify SCA permettendo, tramite un'interfaccia grafica utente, l'organizzazione, l'analisi e l'interpretazione dei risultati.

In particolare, tale strumento software consente di:

- identificare puntualmente le vulnerabilità di sicurezza mostrando il nome del file ed il numero della linea di codice;

- visualizzare il codice vulnerabile, contestualizzandolo rispetto all'intera struttura dell'applicativo;
- identificare i flussi dei dati che sono causa delle problematiche, al fine di una loro migliore comprensione;
- mostrare consigli dettagliati sulle vulnerabilità in modo tale da comprendere come esse possano essere trattate;
- gestire le opzioni di ordinamento, filtraggio ed interrogazione così che possano essere evidenziate specifiche categorie di vulnerabilità o specifiche tipologie di analisi;
- gestire i profili di analisi avanzati tramite il Wizard AuditGuide.

In Figura 3.2 è illustrata l'interfaccia principale del Fortify Audit Workbench.

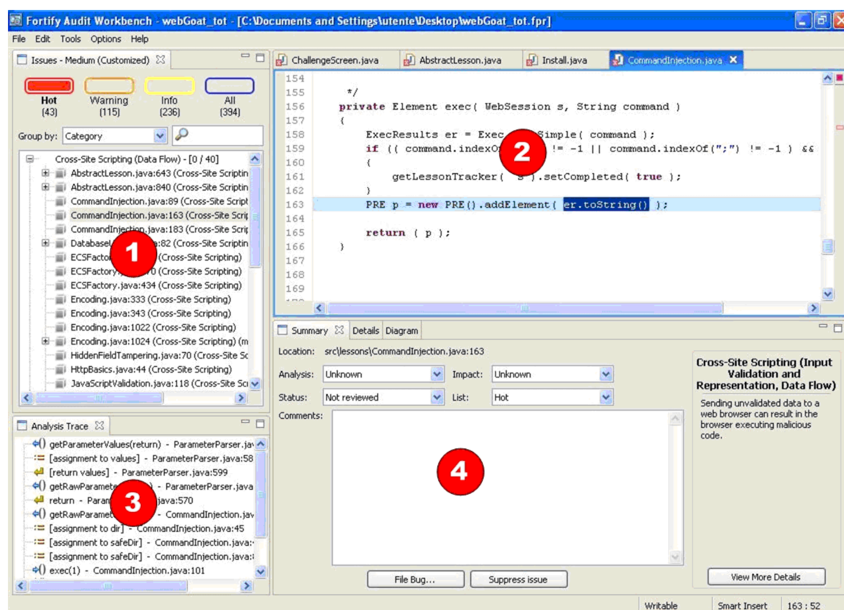


Figura 3.2. Interfaccia principale del Fortify Audit Workbench

In tale interfaccia è possibile distinguere i seguenti quattro riquadri:

1. *Pannello delle problematiche*: mostra le vulnerabilità individuate e ne consente l'ordinamento ed il raggruppamento in base a criticità, categoria, problematica rilevata e file interessato.
2. *Visualizzazione del codice sorgente*: fornisce una visualizzazione delle linee di codice all'interno delle quali è stata individuata la problematica. Al suo interno possono essere aperti contemporaneamente più file; in questo modo

è possibile seguire, in combinazione con il pannello di tracciamento, l'intero "percorso" della problematica.

3. *Pannello di tracciamento*: fornisce tutte le informazioni relative alle componenti, alle variabili ed alle funzioni coinvolte nella problematica selezionata all'interno del pannello delle problematiche. L'analisi di tali informazioni consente una comprensione della problematica migliore e più approfondita.
4. *Area di informazioni sulla vulnerabilità*; quest'area, a sua volta, è suddivisa in:
   - *Summary*: visualizza sommariamente i risultati di analisi;
   - *Details*: mostra puntualmente i dettagli e le informazioni sulla problematica selezionata;
   - *Diagram*: illustra il diagramma di flusso delle componenti dell'applicativo coinvolte nella problematica selezionata.

L'importazione dell'analisi grezza avviene attraverso l'apertura del file con estensione `.fpr` da parte di Fortify Audit Workbench. Qualora fosse possibile, prima di effettuare tale operazione, si dovrebbe rendere disponibile, sul sistema dove la fase di analisi preliminare avrà luogo, il codice sorgente relativo al file `.fpr` da esaminare. In assenza di tale codice sorgente completo, saranno mostrate nello schermo esclusivamente le linee di codice strettamente connesse alle problematiche segnalate.

La selezione delle firme di rilevamento avviene tramite il Rulepack Management, strumento raggiungibile attraverso la voce "Manage Rulepacks", presente all'interno del sottomenù "Tools".

All'interno della finestra associata al Rulepack Management è possibile effettuare la scelta del livello di sicurezza (Broad, Medium o Targeted) da assegnare ad ogni pacchetto di firme. La stessa operazione può essere effettuata su tutti i pacchetti di firme selezionando, dal menù a tendina "Select Rulepack" la voce "All Rulepacks".

Ne consegue che una prima attività di selezione deve avvenire ad alto livello, ovvero sulla base delle seguenti opzioni:

- *Broad*: tale impostazione è volta a comprendere l'intero catalogo di firme di rilevamento disponibili.
- *Medium*: tale impostazione è volta a garantire un bilanciamento tra la produzione di risultati che dettaglino ogni potenziale problematica e la limitazione del "rumore".
- *Targeted*: tale impostazione è volta a comprendere quelle firme individuate come problematiche ad alta priorità, in particolare rispetto al contesto della specifica sessione di analisi.

Nel caso in cui dovessero esserci specifiche esigenze di analisi e verifica, è possibile effettuare una selezione personalizzata delle voci presenti nel pacchetto di firme; ciò avviene cliccando sul pulsante "Customize", raggiungendo la voce

interessata e selezionando o deselezionando la stessa tramite la corrispondente checkbox.

In seguito alla conferma dell'avvenuta selezione, è possibile notare la modifica del numero di occorrenze presenti nel pannello "Issues". Le problematiche sono classificate in base ad una scala di tre livelli di criticità, di gravità decrescente; tali livelli sono:

- *Hot*: problematiche ad alta criticità;
- *Warning*: problematiche a media criticità;
- *Info*: problematiche a bassa criticità.

La classificazione all'interno di tali livelli di criticità avviene in base ad una valutazione, strettamente tecnologica, del possibile impatto della problematica segnalata.

Dopo aver controllato il risultato derivante dall'attuale selezione di firme, è possibile creare ed esportare il relativo filtro che conterrà al suo interno la configurazione dell'analisi preliminare.

Tale file potrà essere successivamente utilizzato in fase di scansione con Fortify SCA specificando l'opzione `-filter` seguita dal nome del file di filtro:

```
sourceanalyzer -b <build-id> -scan -f result.fpr -filter filtro.txt
```

Infine, è possibile salvare il progetto utilizzando la voce "Save Project As" presente all'interno del sottomenù "File". È buona pratica specificare un nome diverso per ciascuna fase di analisi così da rendere possibile una precisa ricostruzione delle diverse fasi di interpretazione e configurazione.

3.2.3 Fortify Manager

Fortify Manager è la "scrivania virtuale" di sicurezza alla quale attingono sia i team di sviluppo sia quelli di sicurezza.

Esso si basa su un'interfaccia Web, eseguita su un Apache Tomcat Application Server, e integra un database HSQL Java; si consiglia l'uso di quest'ultimo solo a scopo di valutazione del software.

Fortify Manager permette resoconti flessibili, centralizza le firme di rilevamento delle vulnerabilità nonché le politiche e la gestione degli avvisi facilitando la revisione di un gran numero di progetti software.

Più specificatamente, Fortify Manager:

- *Centralizza la gestione*; a tal fine esso:
  - consente ai controllori della sicurezza la gestione di molteplici revisioni di progetti interagendo da un solo terminale;
  - permette la configurazione ed il monitoraggio delle politiche di sicurezza di tutti i progetti software soggetti a revisione;

  - genera automaticamente resoconti rivolti ai team di gestione, di sicurezza e di sviluppo;
  - attenua i rischi associati allo sviluppo esterno del software applicando le più idonee misure di controllo al codice fornito.
- *Organizza secondo un ordine di priorità*; più specificatamente, esso mette in evidenza le vulnerabilità più gravi sfruttando l'alta configurabilità degli applicativi che lo compongono; in aggiunta ad una migliore gestione della sicurezza, esso evita efficacemente inutili pressioni sulle linee di sviluppo dovute ai risultati delle analisi.
- *Evidenzia le tendenze*; le statistiche effettuate sui dati grezzi permettono ai team di sicurezza di identificare e documentare le differenze sostanziali che si riscontrano tra le differenti analisi; ciò permette di sottolineare più efficacemente i cambiamenti più significativi relativi ai risultati delle analisi.
- *Mette in guardia*; appena viene identificata una nuova vulnerabilità esso emette gli opportuni avvisi, rendendo superflua la revisione continua e integrale dei risultati di un'analisi; ciò implica un risparmio di tempo sulla revisione dei risultati, garantendo una maggiore dedizione all'analisi ed alla comunicazione dei rischi e dei rimedi.

Fortify Manager costituisce anche un'importante risorsa di dati; questi vengono immagazzinati durante lo sviluppo in sicurezza del software e forniscono a tutti i committenti le necessarie informazioni di sicurezza in tempo reale. I dati immagazzinati possono includere i progetti, i risultati delle analisi, i dettagli delle revisioni, le firme di rilevamento delle vulnerabilità, i vari tipi di politiche di sicurezza e le identità degli utenti.

Fortify Manager sottolinea le relazioni esistenti tra informazioni di varia natura permettendo alle organizzazioni di tenerne conto proficuamente.

Fortify Manager prevede cinque profili utente; essi sono:

- *Amministratore*:
  - gode di tutti i privilegi e, pertanto, può eseguire ogni tipo di operazione;
  - può accedere all'intero pannello di amministrazione; quest'ultimo include i menù relativi alla gestione dei pacchetti di firme di rilevamento, all'organizzazione, alla personalizzazione e agli utenti.
- *Direttore*:
  - può creare e gestire progetti e gruppi di progetti;
  - può creare e gestire politiche basate su eventi specifici che riguardano chiunque sia interessato ad un progetto;
  - può creare politiche di "sign-off";
  - può creare e gestire gruppi di utenti; gode degli stessi privilegi dello Sviluppatore.
- *Sviluppatore*:

 - può visualizzare l'interfaccia principale, i progetti, i resoconti delle analisi del codice, le vulnerabilità, lo stato delle politiche e frammenti di codice;
 - può trasferire i risultati delle analisi nel database di Fortify Manager;
 - può creare e modificare le politiche personali legate agli eventi e agli avvisi;
 - può generare report.
- *Revisionatore*: gode degli stessi privilegi dello Sviluppatore ad eccezione del caricamento dei risultati dell'analisi e della visione di frammenti di codice.
- *Utente remoto*: può soltanto trasferire in Fortify Manager i risultati dell'analisi tramite appositi strumenti software remoti; non può accedere all'interfaccia Web di Fortify Manager.

In Figura 3.3 sono riportati i risultati di un'analisi visualizzati tramite Fortify Manager.

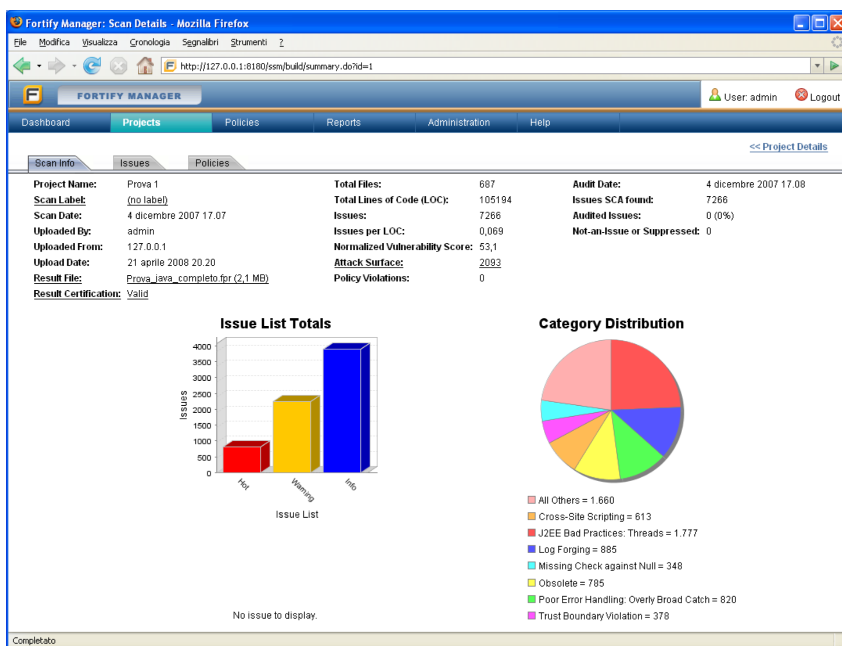


Figura 3.3. Risultati di un'analisi tramite Fortify Manager

3.2.4 Secure Coding Rulepacks

Le *firme di rilevamento* rappresentano le definizioni necessarie per identificare gli elementi del codice sorgente che possono causare problemi di sicurezza.

Fortify Software integra nella propria suite dei pacchetti di firme di rilevamento che costituiscono il Secure Coding Rulepacks. Quest'ultimo è il frutto di anni di esperienza nell'ambito della sicurezza del software e costituisce una ricca riserva di informazioni circa librerie e pratiche di programmazione comunemente utilizzate nello sviluppo del software; esso rappresenta un bagaglio di conoscenza che cresce in dimensione e qualità grazie all'impegno degli esperti del settore.

Ogni pacchetto di firme comprende un vasto numero di regole e ogni regola definisce un particolare comportamento anomalo da rilevare nel codice sorgente sottoposto ad analisi.

Quando vengono individuate le vulnerabilità, i pacchetti di regole forniscono informazioni dettagliate su di esse così che gli sviluppatori possano impiegare il loro tempo nell'elaborazione e nell'implementazione delle correzioni piuttosto che nella ricerca di dettagli di sicurezza relativi a tali criticità. Le informazioni fornite riguardano specifiche nozioni sulla categoria della vulnerabilità, su come possa essere sfruttata da un attaccante e su come gli sviluppatori possano rendere il loro codice immune da questi attacchi.

Secure Coding Rulepacks supporta vari linguaggi, quali .NET, C/C++, ColdFusion 5.0, Java, SQL, e le rispettive estensioni.

Fortify Source Code Analyzer utilizza Secure Coding Rulepacks come conoscenza di base per effettuare l'analisi.

Nelle Sezioni 6.3 e 7.5 sono riportate rispettivamente le vulnerabilità dei linguaggi C/C++ e Java individuate da Fortify SCA e da Secure Coding Rulepacks.

3.2.5 Rules Builder

Nelle loro varie forme, le firme di rilevamento permettono, ad esempio, di individuare quali funzioni siano intrinsecamente insicure oppure in quale cono o specifica sequenza di programma lo diventino.

I pacchetti di regole messi a disposizione da Secure Coding Rulepacks rilevano migliaia di costrutti di codice vulnerabili e i possibili pericoli nell'utilizzo dei dati.

Ciò nonostante potrebbe essere necessario estendere le funzionalità di Fortify SCA o di Secure Coding Rulepacks creando delle firme di rilevamento personalizzate tramite Rules Builder.

Infatti, ci si potrebbe trovare nelle condizioni di dover rispettare le stringenti linee guida di sicurezza di un'organizzazione o di dover analizzare un progetto che utilizza librerie o codici binari pre-compilati di terze parti che non sono compresi in Secure Coding Rulepacks.

Fortify SCA utilizza due tipi di firme di rilevamento sviluppate dal Fortify Security Research Group. La prima famiglia di firme è relativa al tipo analizzatore utilizzato e comprende le seguenti firme: Semantic Rules, Data Flow Rules, Control Flow Rules, Configuration Rules e Structural Rules.

La seconda famiglia di firme è relativa al codice sorgente; tali firme operano indipendentemente da Fortify SCA; esse sono: Alias Rules, Allocation Rules, Buffer Copy Rules, Non-Returning Rules e String Length Rules.

Parte II

Progettazione delle attività di code inspection

Questa seconda parte presenterà il contesto applicativo di riferimento per la presente tesi; essa è strutturata in quattro capitoli. Il Capitolo 4 illustrerà il contesto aziendale rispetto al quale si è concentrata la nostra attività. Il Capitolo 5 presenterà l'insieme delle operazioni di Code Inspection richieste, introducendo il lettore alla tecnologia e alla metodologia utilizzate. Nel Capitolo 6 verrà illustrata una serie di "best practices" per lo sviluppo di codice sicuro C/C++, riconosciute dai massimi esperti a livello internazionale. Nel Capitolo 7 verrà illustrata una serie di "best practices" per lo sviluppo di codice sicuro Java, riconosciute ufficialmente da Sun.

4

Descrizione della realtà di riferimento

Obiettivo principale di questo capitolo è quello di presentare il contesto aziendale nel quale si è svolta l'attività di tesi. Particolare attenzione verrà dedicata agli aspetti legati alla sicurezza che hanno costituito l'elemento cardine di ogni nostra campagna sperimentale. Gran parte del capitolo sarà dedicata all'introduzione del lettore alle procedure operative che caratterizzano il processo di code inspection nella realtà di riferimento.

4.1 Premessa

Il contesto aziendale di riferimento, è quello di una grossa azienda di telecomunicazioni che risulta essere un cliente storico di ACSI Informatica. La società in questione è leader nell'ingegneria e nella realizzazione di reti e sistemi di telecomunicazioni. Per questioni commerciali non possiamo specificare il nome della società; per tale ragione, nel seguito della tesi, useremo lo pseudonimo "TLCompany" per indicarla.

Il nostro settore di interesse è stato quello della sicurezza informatica. In questo senso TLCompany fornisce ormai da anni soluzioni di elevato contenuto tecnologico a diverse società, di carattere nazionale e internazionale, appartenenti ai diversi segmenti delle telecomunicazioni e del settore ICT (Information and Communication Technology).

L'opportunità principale, di fronte ad un contesto relazionale di questo tipo, è stata quella di rapportare, in maniera diretta, il lavoro svolto sulla sicurezza informatica e, in particolare, sul processo di code inspection, con i più importanti strumenti a disposizione sul mercato (primo fra tutti il software Fortify SCAS introdotto nel Capitolo 3) e con delle vere e proprie attività di tipico business aziendale.

Nella sezione successiva presenteremo il contesto di riferimento in cui hanno avuto luogo le attività sperimentali.

Esso è riferito al processo di code inspection, all'interno di un contesto di protezione di attività di business, gestito da TLCompany per conto di una società di telefonia mobile di estrema rilevanza nazionale, su determinate aeree del territorio italiano.

È necessario, però, evidenziare come, trattandosi di un contesto estremamente sensibile, il cui elemento portante è, appunto, la "sicurezza informatica", ed essendo basato su dati aziendali riservati, ci siamo dovuti limitare ad effettuare una presentazione che risultasse opportunamente "censurata", oscurando quei fattori critici che potrebbero risultare determinanti contro la sicurezza dei sistemi cardine o che potrebbero violare determinati requisiti di riservatezza aziendale.

4.2 Ambiente di riferimento

Per poter far fronte alla particolare fluidità delle offerte dettata dal mercato e dalla concorrenza, il modello organizzativo di riferimento prevede un fortissimo ricorso all'esterno per lo sviluppo di tutte le applicazioni sulle quali si basa il business aziendale.

L'adozione di questa scelta è ampiamente motivata da un insieme di diversi fattori tra i quali il ridottissimo time-to-market caratteristico del mondo della telefonia mobile, la necessità di dover ricorrere ad una moltitudine di competenze specialistiche verticali nei diversi settori coinvolti, la rapida e continua evoluzione dei linguaggi e paradigmi di programmazione, la presenza diffusa di librerie, oggetti e protocolli proprietari o coperti da brevetto, oltre al bisogno di mantenere un'elevata flessibilità rispetto alla domanda.

Gli eseguibili prodotti dai fornitori in base alle specifiche e ai requisiti forniti dai responsabili di progetto, e successivamente contestualizzati dai capi progetto tecnici, vengono attualmente rilasciati in esercizio a valle di una serie di collaudi funzionali e prestazionali, senza tuttavia subire nessuna forma di controllo o validazione di qualità o di sicurezza, lasciando, di fatto, gli sviluppatori liberi di scegliere i metodi e le forme di programmazione atti a raggiungere i risultati richiesti.

In questo modo l'azienda si assume un rischio elevato legato alla possibile presenza sugli applicativi in produzione di vulnerabilità o, addirittura, di potenziali frodi, sfruttabili da malintenzionati casuali o organizzati.

In quest'ottica è sembrata necessaria l'introduzione, all'interno del processo di sviluppo, rilascio, manutenzione e manutenzione evolutiva degli applicativi, di meccanismi di analisi e validazione del software, sia dal punto di vista della sicurezza sia in ottica anti-frode.

4.2.1 Cenni sul processo di sviluppo delle applicazioni

In questa sottosezione verranno sinteticamente illustrati gli elementi principali del processo di produzione ed aggiornamento delle applicazioni della società di telefonia mobile di nostro interesse, in modo da derivarne dei requisiti tecnologici ed organizzativi di alto livello che orienteranno le scelte sulle modalità d'ispezione.

Procediamo con la descrizione concentrandoci sulle parti del ciclo di vita dell'applicazione utili alla descrizione delle nostre attività, tralasciando le pur fondamentali fasi di creazione della domanda da parte del cliente interno, di formazione della proposta d'investimento o del profilo d'offerta, nonché della dismissione, con conseguente eliminazione dei dati, di un'applicazione.

Consideriamo, quindi, il caso della *manutenzione evolutiva*, in quanto esso risulta analogo a quello di uno sviluppo ex-novo pur essendo di gran lunga il più diffuso in quanto ad occorrenze.

A fronte di una nuova esigenza, a prescindere da come questa sia scaturita, il project manager provvede a coinvolgere i responsabili delle applicazioni interessate dal servizio; questi recepiscono, integrano e formalizzano i requisiti per richiedere ai fornitori, secondo le modalità e i tempi concordati, la consegna del prodotto, insieme con la sua documentazione e manualistica, con il codice sorgente, con i parametri di compilazione e, soprattutto, con il suo eseguibile. Quest'ultimo, vero elemento fondamentale, verrà, con la collaborazione dei suoi sviluppatori, installato prima in collaudo per subire le prove funzionali e prestazionali in modo da verificarne la rispondenza ai requisiti e, successivamente, anche a valle di eventuali ulteriori ritocchi e messe a punto, verrà posto in esercizio per l'avvio della fase di produzione.

Falle casuali o dolose nel codice dell'applicativo, oltre che praticamente irrilevabili, si ripercuotono in profondità sulla logica di funzionamento dell'applicativo pregiudicando anche gravemente sia il patrimonio informativo sia la stessa capacità di erogazione del servizio.

In particolar modo in applicativi orientati ai servizi, tali falle potrebbero, inoltre, interessare altre applicazioni o aree contigue all'applicazione o con le quali questa si trova a dover comunicare, o comunque interagire, via middleware o altri canali di trasferimento in sistemi consolidati o virtuali.

4.2.2 Volumi in gioco

Il processo industriale di sviluppo degli applicativi nell'ambito di nostra pertinenza segue quattro cicli di rilascio annui che prevedono l'applicazione dei nuovi eseguibili in produzione sotto forma di kit. Per ogni applicativo, pertanto, all'atto del suo commissionamento, deve essere specificato per quale dei quattro kit trimestrali dell'anno esso dovrà essere rilasciato. Il numero complessivo di singole applicazioni presenti sul piano di sicurezza generale ammonta a 301 unità.

4.2.3 Caratteristiche particolari

Puntiamo adesso l'attenzione sui rischi e sulle minacce impliciti specifici partendo dalle caratteristiche peculiari legate allo sviluppo delle applicazioni nell'ambiente di riferimento. Di seguito vengono elencati gli aspetti inerenti allo sviluppo delle applicazioni maggiormente rilevanti dal punto di vista della sicurezza.

- *La maggior parte delle applicazioni sono sviluppate ad-hoc*: non trattandosi di prodotti commerciali off-the-shelf disponibili per tutti, non vi è la possibilità di reperire l'esperienza di altri utilizzatori o di ricercatori nel campo della sicurezza.
- *Le applicazioni sono numerose*: vi è l'oggettiva difficoltà di potersi concentrare su di un numero limitato di applicazioni.
- *Le applicazioni vengono continuamente aggiornate per inserirvi nuove funzioni e per adattarsi alle esigenze dettate dal mercato*: ogni nuova versione inserita nel kit è praticamente da considerarsi alla stregua di una nuova applicazione.
- *Le applicazioni vengono sviluppate in tempi brevi e stringenti per rispettare il ridotto time-to-market*: i tempi di sviluppo ristretti potrebbero influire sulla qualità del prodotto rilasciato a scapito, soprattutto, della sicurezza.
- *Le applicazioni vengono sviluppate da fornitori diversi, nella maggioranza dei casi in outsourcing*: questi fornitori potrebbero avere sensibilità differenti rispetto alla sicurezza.
- *Su singole applicazioni si appoggiano servizi diversi*: una vulnerabilità potrebbe emergere solo a fronte di un particolare servizio, oppure, potrebbe ricadere su più servizi contemporaneamente.
- *Una parte consistente delle applicazioni tratta dati sensibili*: ciò comporta, da una parte, una maggiore appetibilità da parte di malintenzionati e, dall'altra, particolari obblighi nei confronti di clienti e dell'Autorità Garante per il Trattamento dei Dati Personali.
- *Molte applicazioni devono necessariamente interagire le une con le altre*: una vulnerabilità su una di esse potrebbe avere ricadute su un'intera catena di business.
- *La maggior parte delle applicazioni deve interagire con utenti, anche esterni*: l'esposizione alle minacce aumenta considerevolmente.

Proseguiamo nella descrizione indicando alcuni dati sulla distribuzione delle applicazioni rispetto ai diversi livelli di criticità.

Su un totale di 315 applicazioni censite, il livello di criticità classificato a valle dell'analisi del rischio, secondo la metodologia in vigore, ha portato a concludere che il 34,6% ha presentato una criticità alta, il 49,5% una criticità media e il rimanente 15,9% una criticità bassa.

La quota di applicazioni che trattano dati personali, ai sensi del D.lgs. 196/2003, equivale a circa il 35% dell'ammontare complessivo.

In particolare, ispirandosi alle normative aziendali in materia, un'applicazione sicura:

- non deve fornire più informazioni di quanto previsto;
- non deve danneggiare altre applicazioni o gli altri dati sui quali essa si appoggia;
- non deve danneggiare le postazioni degli utenti, siano essi interni o esterni all'azienda;
- non deve essere in grado di danneggiare o portare attacchi nei confronti di sistemi o reti, siano essi interni o esterni;
- non deve consentire a nessuno di interferire con la sua logica di funzionamento;
- non deve consentire ad utenti o sistemi di impersonare altri utenti o sistemi;
- deve essere in grado di tracciare tutte le anomalie o gli eventi rilevanti, dal punto di vista della sicurezza, che essa incontra;
- dovrebbe essere in grado di segnalare allarmi a fronte di eventi significativi che essa riscontra.

4.3 Strutturazione del processo di code inspection nell'ambito di riferimento

L'attività di ispezione del codice sorgente gestita da TLCompany è realizzata con l'obiettivo di individuare vulnerabilità significative ai fini della sicurezza al fine di impedire che esse possano essere sfruttate per alterare il normale comportamento dell'applicazione in esame.

In particolare, l'attività realizza i seguenti obiettivi:

- analisi delle informazioni che caratterizzano l'applicazione in esame dal punto di vista funzionale e individuazione delle sue aree di esposizione;
- estrazione delle vulnerabilità tecnologiche rilevate;
- classificazione delle vulnerabilità riscontrate all'interno del contesto aziendale dell'applicazione;
- produzione del "Rapporto di Verifica" per l'applicazione analizzata.

La descrizione del flusso da seguire durante lo svolgimento dell'attività di ispezione del codice sorgente, insieme alle indicazioni necessarie per poter comprendere i diversi passaggi e ai criteri da adottare per operare le scelte corrette saranno descritti nel prosieguo.

Per poter svolgere tali attività è necessario procedere seguendo un percorso strutturato in sette fasi, distinte e successive, all'interno delle quali verranno effettuate le operazioni che porteranno man mano a restringere il campo di osservazione agli aspetti peculiari dell'applicazione sotto esame e a fornire gli elementi necessari ad interpretare le vulnerabilità rilevate, consentendone una valutazione oggettiva e focalizzata sulle principali funzioni da questa utilizzati.

Tali fasi sono di seguito elencate:

1. verifica formale della completezza del materiale e della documentazione;
2. caricamento del materiale sulla piattaforma di analisi;
3. individuazione delle aree di esposizione dell'applicazione;
4. selezione delle firme di rilevamento;
5. identificazione delle occorrenze e classificazione delle vulnerabilità tecnologiche rilevate;
6. contestualizzazione delle vulnerabilità;
7. organizzazione dei risultati.

Nella Figura 4.1 si riporta il flusso primario del processo di code inspection. Al suo interno sono riportate le componenti ed i passi fondamentali del processo di individuazione e analisi delle problematiche di sicurezza.

Si noti come le fasi possano essere considerate soltanto quattro, qualora si considerino i passi 3 e 4 facenti parte dell'Analisi Preliminare ed i rimanenti passi 5, 6 e 7 facenti parte dell'Analisi Avanzata. La gestione dell'output può considerarsi una fase successiva al processo di ispezione del codice sorgente vero e proprio.

L'instaurazione di un primo livello di analisi è legata alla necessità di effettuare un raffinamento dei risultati ottenuti, eliminando tutto ciò che può considerarsi "rumore". Tale processo consente di giungere ad una configurazione delle regole di ispezione tale da potersi considerare sufficiente alla prosecuzione delle attività.

Gli obiettivi primari di un livello di analisi di questo tipo, sulla base delle precedenti considerazioni, si possono classificare in:

- esecuzione di un processo di "purificazione" dei risultati (ad esempio, attraverso l'eliminazione di falsi positivi);
- ottenimento di una configurazione finale delle regole di ispezione ritenute opportune ai fini del contesto applicativo.

Il secondo livello ha lo scopo di ottenere un'analisi maggiormente approfondita delle vulnerabilità individuate sul codice sorgente per la successiva emissione dei risultati. I dati ottenuti dalla reiterazione della prima fase di analisi devono essere posti possibilmente, in relazione tra loro al fine di identificare eventuali collegamenti o dipendenze. L'analisi delle informazioni ottenute consente il raffinamento dei risultati e, soprattutto, di definire quale tipologia o metodologia di intervento debba essere specificatamente applicata.

Nelle sottosezioni seguenti verranno descritte le operazioni legate a ciascuno dei passi indicati.

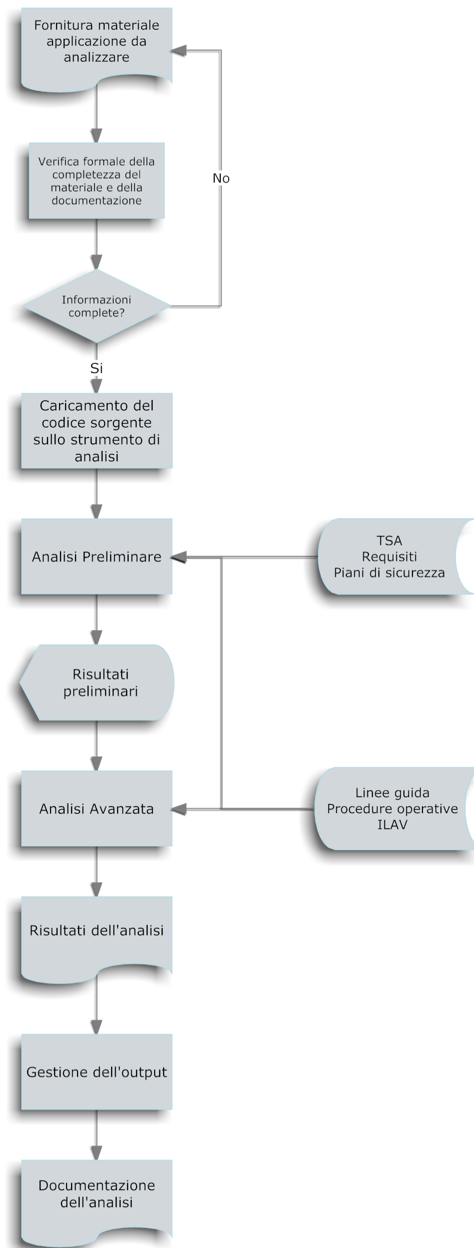


Figura 4.1. Processo di Code Inspection

4.3.1 Verifica formale della completezza del materiale e della documentazione

L'ottenimento di un insieme di elementi, codici ed informazioni, indispensabili per poter svolgere le attività di ispezione, è un passo preliminare fondamentale.

Sarà, pertanto, necessario assicurarsi che sia stato messo a disposizione tutto quanto necessario per poter procedere con i passi successivi.

È da notare che, durante questa fase, la verifica del materiale ricevuto avviene in maniera prettamente formale e non sostanziale, dal momento che, in questo momento dell'attività, non è ancora possibile valutarne il contenuto e, soprattutto, la completezza.

Infatti, per esempio, il codice ottenuto potrebbe a sua volta far riferimento ad altre porzioni di codice necessarie ai fini dell'analisi ma assenti al momento della consegna. Tale mancanza non potrà essere evidenziata prima del caricamento all'interno della piattaforma di analisi. D'altra parte, il codice mancante potrebbe, a sua volta, richiamare altre porzioni, e così via attraverso diverse iterazioni. Da questo esempio si capisce l'importanza di ottenere in una volta sola tutto quanto necessario alla compilazione, insieme alle relative istruzioni.

Un altro esempio consiste in una richiesta di descrizione approfondita di un flusso informativo, o di una particolare funzione, che all'inizio dell'analisi sembrava sufficientemente illustrato nella descrizione dell'architettura, ma che, sulla base di riscontri positivi di particolari vulnerabilità, potrebbero dover essere studiati specificatamente nel loro contenuto al fine di validare la vulnerabilità tecnologica e poterla classificare correttamente.

Informazioni necessarie all'analisi

Le informazioni minime necessarie per poter iniziare l'ispezione del codice sorgente di un'applicazione possono essere riassunte nelle voci seguenti:

- codice sorgente completo ed eventuali librerie;
- istruzioni per la compilazione;
- descrizione dell'architettura dell'applicazione (comprensiva delle interazioni interne ed esterne, della loro modalità e delle categorie di macro-dati impiegate);
- "Piano di Sicurezza" dell'applicazione (se disponibile);
- "Technical Security Audit" (se disponibile).

Lo studio del Piano di Sicurezza, se disponibile, potrebbe già di per sé fornire gran parte delle informazioni necessarie, con il vantaggio che, al suo interno, molti degli aspetti da trattare sono già stati considerati e vagliati dal punto di vista della sicurezza.

Inoltre, la consultazione dei risultati di Technical Security Audit (nel seguito, TSA) che dovessero essere stati eseguiti nei confronti dell'applicazione in esercizio

per la quale si vuole effettuare l'analisi, consentirebbe, da una parte, di individuare con precisione le cause che hanno portato al manifestarsi di vulnerabilità e, quindi, di consentire la loro eliminazione, dall'altra di capire il suo comportamento verso l'esterno, permettendo di ottenere una definizione più precisa delle aree di esposizione.

Il *Modulo di Richiesta Analisi* deve essere compilato dalla Linea richiedente in base all'apposito template di riferimento. In particolare, si deve verificare che siano correttamente riportate e complete tutte le informazioni della Tabella 4.1; questa specifica, appunto, tutte le informazioni necessarie alle attività di code inspection. Per ciascuna tipologia di informazione viene indicata, se possibile, la fonte principale di provenienza.

Solo dopo aver verificato la completezza e la correttezza delle informazioni ottenute, e delle eventuali integrazioni richieste, si potrà passare alla fase di acquisizione del codice.

Informazioni	*Note*	*Fonte primaria*
Nome dell'applicazione e sua versione	Nome dell'Applicativo e degli eventuali suoi moduli da sottoporre a Code Inspection	PDS/Linea Sviluppo
Criticità dell'applicazione	C1/C2/C3	PDS/Linea Sviluppo
Categoria dell'informazione	Tipologia di informazioni gestita, sfruttata o memorizzata dall'applicazione	PDS/Linea Sviluppo
Stato di deployment	Informazione circa il rilascio o meno dell'applicazione	PDS/Linea Sviluppo
Risultati precedente analisi	Se esistenti, i risultati di analisi precedenti saranno confrontati con i nuovi	Gruppo di Code Inspection
Risultati TSA	Informazioni provenienti dal gruppo TSA relative allo sviluppo	Gruppo Verifica (TSA)
Informazioni PDR	Informazioni provenienti dai Piani di Rientro, se relative allo sviluppo	PDR
Architettura e moduli dell'applicazione	Logica dell'applicazione	PDS/Linea Sviluppo
Applicazioni esterne	Informazioni circa applicazioni esterne che saranno, o potrebbero essere, utilizzate	PDS/Linea Sviluppo
Interfacce dell'applicazione	Informazioni circa le interazioni dell'applicazione con il "mondo esterno"	PDS/Linea Sviluppo
Deployment Framework (con informazioni sulla versione)	Specifica se e quale framework verrà utilizzato per il deployment dell'applicativo	Linea Sviluppo
Deployment Framework (dettagli di configurazione)	Configurazione dettagliata del framework di deployment	Linea Sviluppo
Protocolli, versione e loro definizione	Definizioni delle versioni dei protocolli, con particolare riferimento a protocolli proprietari o personalizzazioni di protocolli standard.	PDS/Linea Sviluppo
Tipologie di risorse utilizzate (File system, Dbms, Network, Web Server, ecc).	Informazioni circa le risorse a cui avrà accesso l'applicazione o le risorse che la stessa richiede	PDS/Linea Sviluppo

Informazioni	*Note*	*Fonte primaria*
Sistemi Operativi	Sistemi operativi utilizzati per lo sviluppo e l'esercizio dell'applicativo.	PDS/Linea Sviluppo
Linguaggi utilizzati (versioni)	Linguaggio/variante/versione utilizzata	PDS/Linea Sviluppo
Compilatori utilizzati	Compilatori, e loro versioni, utilizzati per la compilazione	PDS/Linea Sviluppo
Librerie (di sistema, esterne o personalizzate)	Tutte le librerie di cui l'applicazione necessita. Le librerie personalizzate potrebbero necessitare di C.I. dedicate	PDS/Linea Sviluppo
Argomenti di Build	Argomenti utilizzati nella compilazione	Linea di Sviluppo
Tecnologia di Build	Ad esempio, gcc/maven/jasper/weblogic/javac/cs	Linea di Sviluppo
Ulteriori specifiche per la compilazione	Informazioni circa la possibilità che siano necessarie delle ulteriori informazioni affinché l'applicazione possa essere compilata	Linea di Sviluppo
Elenco completo dei file da sottoporre a C.I.	Informazioni puntuali relativamente al codice sorgente da sottoporre ad Analisi. Per ciascun file deve essere indicato il percorso relativo completo e l'hash SHA1	Linea di Sviluppo

Tabella 4.1. Informazioni necessarie all'analisi

Punti di attenzione

Benché tutte le informazioni presentate in Tabella 4.1 siano da considerarsi necessarie all'effettuazione delle attività descritte nel prosieguo del presente documento, è necessario porre particolare attenzione sulla completezza e sull'aggiornamento di quanto ottenuto.

Di seguito viene posta l'attenzione su quelle tipologie di informazioni la cui assenza o non completezza potrebbero costituire un blocco al prosieguo delle attività di code inspection sul codice sorgente fornito.

Architettura e moduli dell'applicazione

La completezza e la chiarezza delle informazioni ottenute a tale proposito sono fondamentali al fine di consentire una corretta interpretazione e valutazione delle problematiche segnalate in sede di analisi avanzata. In sede di analisi preliminare, inoltre, tali informazioni sono indispensabili al fine di circoscrivere al meglio il contesto all'interno del quale sarà effettuata la code inspection.

Interfacce dell'applicazione

Consentendo il contatto tra l'applicativo sottoposto ad analisi (sia esso un'applicazione Web o un processo che effettua chiamate ad un'API) le interfacce rappresentano tipicamente il punto di maggior esposizione di un applicativo.

Ai fini di una corretta valutazione degli impatti e del rischio, la documentazione ottenuta dovrà contenere una loro puntuale descrizione relativamente alla tipologia e ai metodi.

Deployment framework e sua configurazione

Qualora l'applicativo, per il suo esercizio, si appoggi su specifici framework, dovranno essere ottenute le specifiche delle configurazioni utilizzate. Ad esempio, in presenza di un applicativo J2EE che impieghi Tomcat per l'erogazione del servizio, dovranno essere ottenute le informazioni per la configurazione di quest'ultimo.

Compilatori utilizzati e dipendenze

L'impiego di specifiche versioni di compilatori può introdurre nel binario ottenuto problematiche peculiari, non legate allo specifico codice sorgente.

L'analisi del codice sorgente deve tener conto anche di tale informazione al fine di una corretta valutazione e comprensione delle problematiche segnalate e della conseguente valutazione del rischio tecnologico.

Allo stesso modo, è necessario tener conto delle informazioni relative alle dipendenze da soddisfare in sede di compilazione relativamente alle dipendenze nei confronti di specifiche versioni di librerie esterne o di sistema.

Informazioni necessarie alla compilazione

Al fine di una completa raccolta delle informazioni, è necessario ottenere tutte le informazioni necessarie alla compilazione dell'applicativo. La completezza di tali informazioni è assolutamente imprescindibile ai fini dell'analisi di applicativi realizzati in linguaggio C.

Le successive attività di analisi sono subordinate al completamento con successo delle attività di compilazione del progetto. A tale proposito, oltre alle informazioni descritte precedentemente, devono essere ottenute anche tutte quelle informazioni (linea di comando, parametri, variabili, modalità specifiche di effettuazione) che possano consentire la compilazione dell'applicativo.

Elenco completo dei file da sottoporre a Code Inspection e loro identificazione univoca

In sede di ispezione del codice è necessario poter verificare, in qualsiasi momento, la completezza del codice sorgente sottoposto ad analisi, nonché la corrispondenza tra quanto sottoposto a verifica e quanto originariamente ricevuto.

In particolar modo, per quanto concerne l'integrità dei file di codice sorgente, e la conseguente garanzia di integrità del processo di code Inspection, viene utilizzato un hash univoco generato tramite SHA1. L'hash è una funzione univoca operante in un solo senso (ossia, che non può essere invertita), atta alla trasformazione di un testo di lunghezza arbitraria in una stringa di lunghezza fissa, relativamente limitata. Tale stringa rappresenta una sorta di "impronta digitale" del testo in chiaro o, nel caso delle attività descritte nel presente documento, di uno specifico file. Utilizzando l'algoritmo SHA1 per ciascun file sarà presente un

hash di lunghezza fissa pari a 160bit. Confrontando l'hash ottenuto in sede di richiesta di attività con quello generato a partire dai file ottenuti è possibile rilevare eventuali discrepanze.

Qualora la verifica degli hash (ricevuti e/o generati) evidenziasse mancate corrispondenze e discrepanze sarà necessario procedere ad una richiesta di rettifica ed integrazione da parte del richiedente dell'attività prima di procedere con le successive attività.

In tutti i casi in cui le informazioni ottenute fossero parziali o non sufficientemente dettagliate si dovrà procedere alla richiesta di ulteriori informazioni o chiarimenti. Tali informazioni, inoltre, dovranno essere integrate, in fase di analisi, con una più approfondita valutazione del rischio.

4.3.2 Caricamento del materiale sulla piattaforma di analisi

Una volta verificata la consistenza del materiale ricevuto, si procede all'inserimento dei file all'interno della piattaforma di analisi.

Tale attività potrà richiedere l'effettuazione di adattamento sui formati (file ASCII terminanti secondo formati UNIX o Windows - carriage return line feed - ed eventuali codepage specifiche) oppure ancora sui nomi dei file e sulle loro estensioni.

Lo strumento di analisi, oltre alle istruzioni ad esso corredate, fornirà indicazioni, attraverso determinati messaggi di attenzione o errore, in grado di assistere gli operatori deputati al caricamento nell'esecuzione dei necessari adattamenti e aggiustamenti.

In questa fase potrà emergere l'eventuale mancanza di componenti o porzioni di codice da richiedere agli sviluppatori per una pronta integrazione.

Le metodologie operative utilizzate per l'acquisizione del codice sorgente sono legate allo strumento software Fortify SCAS introdotte nel capitolo precedente ed impiegate più dettagliatamente nei successivi capitoli.

4.3.3 Individuazione delle aree di esposizione dell'applicazione

In base alle peculiarità dell'applicazione sotto esame, è necessario procedere ad un'analisi volta a definire quali aree funzionali andranno osservate. La Tabella 4.2 riporta le macro-categorie di vulnerabilità indicate dalla documentazione aziendale all'interno delle quali sono state riportate le voci di dettaglio più significative.

Sulla base delle caratteristiche dell'applicazione (per esempio, se accede al sottosistema dischi, se è un'applicazione Web, se interagisce con utenti o soltanto con sistemi, se tratta informazioni testuali o effettua calcoli numerici, se impiega funzioni crittografiche o, ancora, se è soggetta a particolari esigenze di tracciamento) verranno selezionate ed evidenziate le macro-categorie, le categorie e le famiglie di vulnerabilità alle quali essa è soggetta ad esposizione.

Macro-Categoria	*Categoria*	*Famiglia*
Validazione dell'input	Script, Servlet e CGI	Shell Execution Command
		File Inclusion
		Cross Site Scripting
		Directory Traversal
		SQL Injection
Bound checking e problematiche di overflow	Stack Overflow	-
	Off-by-one/Off-by-few	-
	Format String	-
	Heap Overflow	Sovrascrivere un puntatore a file
	Integer Overflow ed altri errori logici di programmazione	-
Session management	Session Stealing ed Hjihacking	Cookie
		Token di sessione
	Accesso ad aree non autorizzate	-
Crittografia	Sniffing ed algoritmi crittografici deboli	-
	Brute Forcing	Weak Keys
		Collisioni
	Rainbow Table e Salt Value	-
	Archiviazione insicura	File System Polling
Error e time handling	User Enumeration	-
	Information Disclosure	-
	Directory Listing	-
	Denial Of Service	Deadlocks
	Race Condition	-
	Privilege Escalation e Bypassing dei permessi utente	-
Processi di tracciamento	Errori comuni nei meccanismi di tracciamento	-
	-	Agevolazione delle attività malevole dell'aggressore
	-	Sospensione del servizio o accesso fraudolento
	-	Oscuramento delle attività dell'aggressore

Tabella 4.2. Albero delle vulnerabilità

L'individuazione delle aree di esposizione dell'applicazione dovrà essere riportata all'interno di un documento denominato "Riepilogo delle Aree di Esposizione". In esso dovrà comparire una copia della Tabella 4.2 sulla quale saranno state evidenziate le parti rilevanti per l'applicazione in esame. Per ciascuna delle parti selezionate occorrerà descrivere i motivi che hanno portato alla loro scelta e i riferimenti dai quali si è partiti per l'analisi, indicando i documenti a disposizione, che andranno allegati. Una volta individuate le aree di esposizione secondo il contesto tecnologico e funzionale dell'applicazione da analizzare, si procederà con la selezione delle firme di rilevamento disponibili sullo strumento di ispezione del codice sorgente.

4.3.4 Selezione delle firme di rilevamento

Gli strumenti commerciali di analisi del codice dispongono di decine di migliaia di firme, articolate secondo classificazioni che dipendono dal produttore. Solitamente

ad ogni firma è associato un livello di criticità o sensibilità su scale da tre a cinque valori.

Alcune di queste firme si concentrano sullo stile di programmazione, mentre altre sulla conformità a determinate normative internazionali per garantirne la compatibilità; alcune firme verificano vulnerabilità certe quali, ad esempio, alcuni costrutti di codice certamente sfruttabili, mentre altre segnalano semplicemente che vengono effettuate chiamate esterne.

In breve, il rischio (come per tutti gli strumenti di ausilio di questo tipo in cui alcune forme determinate possono configurarsi come vulnerabilità soltanto in certi casi e non in maniera generale) è di lanciare un'analisi e di ottenere diverse migliaia di occorrenze la maggior parte delle quali classificabili come falsi positivi oppure, escludendo troppe firme, di non rilevare vulnerabilità gravi effettivamente presenti nel codice, ottenendo falsi negativi.

Le firme di rilevamento andranno scelte in modo da adattarsi al particolare piano d'esposizione dell'applicazione, così come esso è stato definito in precedenza.

Di conseguenza, la loro selezione dipende direttamente dalle peculiarità dell'applicazione.

4.3.5 Identificazione delle occorrenze e classificazione delle vulnerabilità tecnologiche rilevate

Durante questa fase si procede all'identificazione e alla categorizzazione dei risultati ottenuti dall'analisi del codice, dopo avere scelto opportunamente le firme da abilitare in base al contesto particolare dell'applicazione sotto esame e al relativo documento "Riepilogativo delle Aree di Esposizione".

Tramite l'applicazione dei passi descritti di seguito si dovranno validare tutti i riscontri emersi e si dovranno classificare gli stessi dal punto di vista dell'effettiva criticità nei confronti dell'applicazione, sia per quanto riguarda l'aspetto tecnologico sia per quanto concerne il contesto.

Questa fase potrà essere reiterata con la precedente in modo da affinarne i risultati e mettere a punto l'insieme di firme più adeguato a ridurre la presenza di falsi positivi.

Studio della firma associata alla vulnerabilità rilevata

Lo studio approfondito e la piena comprensione della potenziale vulnerabilità segnalata dallo strumento sono fondamentali per capire se quanto riportato è effettivamente applicabile al caso specifico, prendendo in considerazione anche le condizioni al contorno, valutandone l'entità e assegnandone la criticità reale di contesto.

Verifica della vulnerabilità all'interno del codice

Una volta compresi i motivi che rendono vulnerabile l'occorrenza rilevata dalla firma, è bene assicurarsi dell'effettiva presenza di tale vulnerabilità all'interno del codice sorgente, verificando scrupolosamente che siano applicabili tutte le condizioni indicate dallo strumento e dalla descrizione della firma associata, seguendo, se necessario, l'articolazione dell'algoritmo, i salti condizionali e tutte le chiamate ad altre funzioni coinvolte nella sezione incriminata.

Valutazione dell'importanza della funzione all'interno della quale è stata rilevata la vulnerabilità

Per riuscire a formulare un valore di criticità per ogni vulnerabilità rilevata occorre valutare l'importanza della funzione all'interno della quale questa è stata trovata rispetto alle funzionalità e al comportamento atteso dell'applicazione.

In questo modo si riuscirà a discriminare tra effettive vulnerabilità che non possono in nessun modo portare danni al funzionamento del sistema (perché non sono esposte o perché agiscono verso sistemi passivi mono-direzionali quali, ad esempio, stampanti o dischi ottici) e quelle vulnerabilità che, invece, espongono l'applicazione e i dati da essa trattati a danni gravi, quali interruzioni, alterazioni del comportamento, modifica non prevista ai dati. Per classificare la gravità delle vulnerabilità effettivamente riscontrate si utilizza la tassonomia riportata nella Tabella 4.3.

4.3.6 Contestualizzazione delle vulnerabilità

I risultati delle precedenti analisi e valutazioni dovranno essere corredati da informazioni aggiuntive al fine di consentire una corretta interpretazione in modo tale da fornire criteri decisionali sulle azioni da intraprendere.

Di seguito vengono indicati i criteri per l'interpretazione dei risultati e la valutazione dello stato dell'applicazione dal punto di vista della sicurezza del suo codice, tenendo in considerazione il suo collocamento all'interno dei processi produttivi e della qualità e sensibilità dei dati che essa tratta o ai quali ha accesso.

A conclusione di questa fase ogni vulnerabilità verrà caratterizzata da un indicatore di criticità aggiuntivo, sempre in riferimento alla Tabella 4.3, in modo da fornire degli elementi di valutazione disgiunti sui due domini di rischio, quello tecnologico e quello di business.

Valutazione della criticità dei dati trattati dalla funzione all'interno della quale è stata rilevata la vulnerabilità

Lo sfruttamento di ogni vulnerabilità che dovesse consentire l'accesso a dati o un'alterazione nella loro elaborazione va valutato sulla base della criticità dei dati stessi.

Broad	*Medium*	*Targeted*
Hot	La vulnerabilità pregiudica la sicurezza dell'intera applicazione e/o dei dati che essa tratta.	Lo sfruttamento della vulnerabilità consente il controllo dell'applicazione e/o l'interruzione del servizio.
Grave	La vulnerabilità potrebbe consentire di variare il comportamento dell'applicazione e di accedere ad informazioni sensibili o a dati riservati e di alterarli.	La vulnerabilità coinvolge i tre parametri di riservatezza, integrità e disponibilità (RID) e consente a chi è in grado di sfruttarla di controllare porzioni dell'applicazione e/o di accedere a porzioni dei dati e di alterarli.
Media	La vulnerabilità potrebbe, in determinate condizioni, portare l'attaccante a conoscenza di informazioni sensibili o di dati riservati.	La vulnerabilità non è immediatamente sfruttabile e coinvolge il solo parametro di riservatezza (R).
Punto d'attenzione	La vulnerabilità non è sfruttabile in nessuno dei modi operativi previsti dall'applicazione.	La vulnerabilità è presente ma risiede in forma latente e non è possibile sfruttarla in nessun modo.
Osservazione	La vulnerabilità è legata alla forma e allo stile del codice o indica una non conformità rispetto ad una normativa internazionale e non implica direttamente la possibilità di essere sfruttata.	La vulnerabilità potrebbe essere legata a stili di scrittura del codice pericolosi perché poco chiari o che potrebbero occultare funzioni nascoste.

Tabella 4.3. Tassonomia delle vulnerabilità

Di conseguenza, una vulnerabilità che dovesse trattare dati di servizio, quali indirizzi IP interni ad un'isola, o parziali, quali gli orari esatti di richieste, senza disporre però della richiesta, va senz'altro considerata molto meno critica rispetto ad una che consenta l'accesso a dati critici di un'applicazione (quali possono essere i dati anagrafici).

La valutazione dei dati trattati dovrà seguire le normative aziendali in vigore, quali i criteri di classificazione e la metodologia di analisi del rischio di dettaglio.

Importanza delle sezioni di codice soggette a vulnerabilità

Le vulnerabilità che compaiono su sezioni dell'applicativo relative a funzioni non esposte, oppure quelle che intervengono su di uno spettro limitato di funzionalità, meno importanti, accessorie o di servizio, sono da ritenersi di gran lunga meno critiche rispetto alle vulnerabilità rilevate su parti principali, quali ingresso e uscita, elaborazione o scrittura dei dati, sempre a seconda dell'applicazione oggetto dell'analisi.

Alcune funzioni, infatti, al di là del fatto che siano o meno esposte verso utenti o sistemi esterni, possono rivestire un'importanza molto limitata all'interno dell'applicazione, equiparabile a funzioni di servizio o, comunque, utilizzate raramente e senza costrizioni sui tempi di esecuzione.

Una vulnerabilità che esponga ad interruzione una funzione di menù utilizzata raramente, senza coinvolgere gli altri thread, sarebbe senz'altro da considerarsi a

criticità bassa rispetto ad una vulnerabilità in grado di arrestare l'esecuzione dell'intera applicazione sul server.

Volume di occorrenze per ogni vulnerabilità rilevata

Se la stessa vulnerabilità ricorre decine o centinaia di volte lungo il listato del codice, potrebbe essere molto importante provvedere a risolverla in quanto, con una singola correzione, si riuscirebbe a migliorare considerevolmente tutto il codice mentre, al contrario, un attaccante che dovesse riuscire a sfruttarla sarebbe in grado di accedere con la stessa tecnica ad una moltitudine di punti d'inserimento all'interno dell'applicazione in esecuzione.

Fermo restando che le diverse sezioni coinvolte da una vulnerabilità di questo tipo potrebbero senz'altro presentare importanze differenti, il fatto che essa sia riscontrabile lungo tutta la catena logica dell'applicazione deve in ogni caso essere preso seriamente in considerazione.

I risultati della contestualizzazione delle vulnerabilità comporranno il documento finale dell'ispezione del codice sorgente, il cosiddetto "Rapporto di Verifica", il quale riassume quanto rilevato e valutato fornendo in modo sintetico i principali problemi riscontrati sulla particolare applicazione considerata.

4.3.7 Organizzazione dei risultati

Durante le diverse fasi che portano all'ispezione del codice sorgente, e che sono state precedentemente descritte, a supporto delle analisi sono state prodotte ed elaborate informazioni relative ad aspetti legati sia all'applicazione in esame sia alla configurazione della piattaforma di analisi.

Tali informazioni sono raggruppate e riportate all'interno dei seguenti documenti semi-lavorati:

- *Riepilogo delle Aree di Esposizione dell'Applicazione*: è il risultato di un'analisi di contesto.
- *Riepilogo di Selezione delle Firme*: è il risultato dell'incrocio delle capacità di analisi dello strumento con le aree di esposizione dell'applicazione.
- *Classificazione delle Occorrenze Rilevate*: è il risultato della validazione e della successiva classificazione delle vulnerabilità dal punto di vista tecnologico.
- *Rapporto di Verifica*: è il risultato della contestualizzazione delle vulnerabilità tecnologiche validate sulla base del contesto di criticità dell'applicazione, dei dati da essa trattati e dei processi di business da essa serviti.

Diversi per livello di dettaglio e d'astrazione, i quattro documenti menzionati affrontano aspetti complementari e sono, comunque, essenziali per poter comprendere i risultati raggiunti nell'analisi di un'applicazione.

Per quanto riguarda il loro raggruppamento, il "Riepilogo delle Aree di Esposizione dell'Applicazione" andrà inserito all'interno del "Rapporto di Verifica", in modo da fornire una descrizione di alto livello delle caratteristiche dell'applicazione dal punto di vista dei suoi rischi, mentre un documento allegato, ovvero il documento "Riepilogo delle Vulnerabilità Riscontrate", dovrà contenere sia l'elenco delle firme selezionate in fase di analisi sia i risultati della validazione e della classificazione delle vulnerabilità, costituendo, così, un preciso ausilio per la risoluzione dei riscontri evidenziati.

Contenuto	*Descrizione*
Nome e caratteristiche dell'applicazione in esame, data dell'analisi e numero univoco di protocollo	Riferimenti alla specifica applicazione in esame, all'ispezione, alla documentazione ricevuta, alla documentazione aggiuntiva eventualmente utilizzata e al materiale ispezionato.
Presenza, numero e descrizione di vulnerabilità critiche	Presenza, numero e descrizione delle vulnerabilità critiche riscontrate a conclusione delle analisi.
Presenza, numero e descrizione di vulnerabilità gravi	Presenza, numero e descrizione delle vulnerabilità gravi riscontrate a conclusione delle analisi.
Presenza, numero e descrizione di vulnerabilità medie	Presenza, numero e descrizione delle vulnerabilità medie riscontrate a conclusione delle analisi.
Presenza numero e descrizione di punti d'attenzione	Presenza, numero e descrizione dei punti d'attenzione riscontrati a conclusione delle analisi.
Presenza, numero e descrizione di osservazioni	Presenza, numero e descrizione delle osservazioni riscontrate a conclusione delle analisi.
Riferimenti	Indicazione di tutti i riferimenti eventualmente impiegati durante le analisi, quali particolari normative, best practice, libri, manuali o altro.
Raccomandazioni	Raccomandazioni conclusive per il responsabile dell'applicazione sulla base di quanto riscontrato a conclusione delle analisi.

Tabella 4.4. Informazioni contenute nel "Rapporto di Verifica"

Contenuto	*Descrizione*
Elenco delle firme che hanno trovato riscontro (per gruppo)	Elenco di tutte le firme selezionate che sono state rilevate dallo strumento, raggruppate per firma.
Classificazione delle firme che hanno trovato riscontro (per gruppo)	Classificazione di tutte le firme selezionate che sono state validate, raggruppate per firma
Elenco completo di tutte le occorrenze	Elenco di tutte le singole firme rilevate dallo strumento, comprensivo del loro riferimento (file, riga).
Elenco delle firme selezionate che non hanno trovato riscontro	Elenco di tutte le firme selezionate che non sono state rilevate dallo strumento.

Tabella 4.5. Informazioni contenute nel "Riepilogo delle Vulnerabilità Riscontrate"

Per comodità si potrà operare direttamente all'interno dei due documenti finali durante le diverse fasi dell'analisi, completando, di volta in volta, le sezioni interessate dall'attività svolta, oppure realizzando quattro documenti distinti semilavorati da far confluire all'interno del "Rapporto di Verifica" e del "Riepilogo delle Vulnerabilità Riscontrate".

Nella Tabella 4.4 sono indicate le informazioni contenute nel "Rapporto di Verifica" mentre nella Tabella 4.5 vengono indicate le informazioni contenute nel "Riepilogo delle Vulnerabilità Riscontrate".

5

Analisi degli interventi di code inspection richiesti

Il presente capitolo, dopo una breve premessa, illustrerà gli interventi di code inspection richiesti nell'ambito di riferimento. Si procederà introducendo sinteticamente la sintassi necessaria per la traduzione dei codici sorgente C/C++ e Java. Successivamente, ci si concentrerà sull'introduzione di passi propedeutici alla realizzazione degli interventi richiesti.

5.1 Premessa

La corretta effettuazione delle attività di analisi è fortemente legata alla puntuale configurazione dell'ambiente, intesa come opportuna scelta dei parametri e delle regole di analisi degli strumenti. Tale attività si basa essenzialmente sulle informazioni ottenute a corredo del codice sorgente a disposizione. L'attenta selezione dell'insieme di regole da utilizzare consente di rendere più efficace il processo di analisi agendo, in prima battuta, sul numero delle possibili vulnerabilità che la successiva fase di consolidamento dei risultati dovrà gestire.

In prima istanza devono essere abilitate tutte quelle regole che, seppur generiche, consentono di verificare la sicurezza ad "ampio spettro" dell'applicativo sottoposto a code inspection. Tale selezione deve essere rifinita nel corso delle reiterazioni che caratterizzano la prima fase di analisi al fine di ridurre, per quanto possibile, la mole di informazioni restituite.

La selezione iniziale delle regole di analisi da utilizzare deve essere effettuata principalmente in base ai criteri che verranno esposti nel corso del presente capitolo.

I requisiti di programmazione e di sicurezza minima individuati a partire dalla documentazione aziendale o identificati in seno al centro di competenza di code inspection costituiscono le direttive che dovranno essere seguite nella selezione delle firme di rilevamento.

L'intero insieme delle regole "custom", eventualmente redatte su precise disposizioni aziendali, dovrà essere sempre e comunque abilitato. È bene sottolineare come la creazione di regole custom possa essere intesa sia come un opportuno processo di modifica applicato a regole preesistenti sia come un vero e proprio processo di realizzazione ex-novo (ad hoc).

5.2 Interventi di code inspection richiesti

La necessità di tenere conto degli specifici requisiti e degli specifici problemi relativi al nostro ambito di riferimento conduce naturalmente alla richiesta di particolari interventi di code inspection. Essi, infatti, consistono nella selezione delle firme di rilevamento dello strumento software utilizzato, cioè Fortify SCAS, che rispecchino le direttive aziendali. La nostra attenzione si concentrerà sui linguaggi di alto livello di più largo impiego, ovvero C/C++ e Java.

A tale scopo, almeno in una fase iniziale, risulta fondamentale giungere ad una certa familiarità con la categorizzazione delle vulnerabilità di sicurezza, tenendo bene in conto le caratteristiche di Fortify SCAS e le tipologie di funzioni a cui tali criticità sono relazionate. Riuscire a sviluppare una sufficiente comprensione di tali tipologie, spesso associate a specifiche criticità, facilita il processo di rilevamento degli obiettivi che conducono ad una opportuna selezione delle regole.

Nel caso in cui fosse possibile, sarebbe opportuno esaminare il comportamento di ogni vulnerabilità e, contemporaneamente, effettuare una valida revisione del codice sorgente o di opportune documentazioni API, così da determinare la peculiarità della regola atta a rappresentare la definizione di un valido comportamento in fase di code inspection.

Per testare la nostra selezione delle firme, verranno sottoposti ad analisi due codici sorgente evidenziandone alcuni aspetti di rilievo e discutendone i risultati ottenuti. Per questioni commerciali non possiamo specificare il vero nome dei codici sorgente; per tale ragione, nel seguito della tesi, per indicare tali codici, useremo gli pseudonimi "Codice 1" e "Codice 2".

Nel seguito, per meglio illustrare gli interventi richiesti, introdurremo la sintassi necessaria per effettuare la traduzione di codici sorgente C/C++ e Java ai fini dell'analisi attraverso Fortify Source Code Analyzer e la metodologia per la selezione delle firme di rilevamento.

5.3 Traduzione di codice C/C++

La sintassi da utilizzare da linea di comando per effettuare la traduzione di un singolo file C/C++ è la seguente:

```
sourceanalyzer -b <build-id> <compiler> [<compiler options>]
```

in cui:

- `<compiler>` è il nome del compilatore che si intende utilizzare durante il processo (ad esempio, `gcc` o `g++`).
- `<compiler_options>` sono le opzioni "passate" al compilatore che vengono tipicamente utilizzate per la compilazione del file.

5.3.1 Integrazione con Make

Fortify SCA consente l'utilizzo del comando `Make` tramite una delle seguenti modalità:

- l'impiego di "Fortify Touchless Build Adapter";
- la modifica di `Makefile` affinchè venga invocato Fortify SCA.

L'analisi di codice sorgente tramite l'adozione di Fortify Touchless Build Adapter utilizza la seguente sintassi di base:

```
sourceanalyzer -b <build-id> Make
```

Nel primo caso, Fortify SCA esegue il comando `Make` senza che `Makefile` venga modificato.

Nel secondo caso, la modifica di `Makefile` consiste nel rimpiazzare ogni chiamata del compilatore o del linker con una chiamata a Fortify SCA. Si tratta di strumenti tipicamente specificati in particolari variabili; ad esempio, se nel `Makefile` originale ci fossero le seguenti dichiarazioni:

```
CC=gcc
CXX=g++
AR=ar
```

esse andrebbero modificate nel seguente modo:

```
CC=sourceanalyzer -b mybuild -c gcc
CXX=sourceanalyzer -b mybuild -c g++
AR=sourceanalyzer -b mybuild -c ar
```

5.4 Traduzione di codice Java

La sintassi da osservare per effettuare la traduzione di un sorgente Java ai fini dell'analisi attraverso Fortify SCA è la seguente:

```
sourceanalyzer -b <build-id> -cp <classpath> <file-list>
```

Nel caso di codice Java, Fortify SCA può sia emulare il compilatore sia accettare file sorgenti in modo diretto. L'emulazione del compilatore avviene nel seguente modo:

```
sourceanalyzer -b <build-id> javac [<compiler options>]
```

Il passaggio in modo diretto avviene rispettando la seguente sintassi:

```
sourceanalyzer -b <build-id> -cp <classpath> [<compiler options>]
\<files>|<file-specifiers>
```

Con `<classpath>` si intende una lista di directory e file jar; nel caso in cui non venga specificato il loro percorso verrà fatto riferimento alla variabile d'ambiente `CLASSPATH`.

Con `<file-specifiers>` si intendono espressioni che consentono di "passare" facilmente una lunga lista di file a Fortify SCA utilizzando metacaratteri. Sono riconosciute due tipologie di metacaratteri, ovvero "*" e "**".

5.4.1 Traduzione di applicazioni J2EE

La traduzione di applicazioni J2EE coinvolge l'analisi di file sorgente Java e componenti J2EE, quali file JSP, descrittori di sviluppo (ad esempio, `web.xml`), e file di configurazione come `struts-config.xml`.

I passi da seguire per effettuare la traduzione sono:

- traduzione di file Java;
- traduzione di file JSP;
- elaborazione dei file di configurazione, tramite la linea di comando:

```
sourceanalyzer -b miobuild-id "miadirectory/miofile.xml"
```

5.5 Attività preliminari

Terminata l'acquisizione del codice sorgente, può partire la fase di visualizzazione dei risultati attraverso Fortify Audit Workbench e, di conseguenza, quella della loro opportuna contestualizzazione sulla base di informazioni che, ad esempio, possono avere carattere architetturale e funzionale o che possono essere strettamente legate al linguaggio utilizzato.

È bene evidenziare che l'obiettivo principale di questa attività è la predisposizione o, ancor meglio, il consolidamento dell'ambiente in previsione della fase di analisi avanzata.

L'attività di raffinamento deve essere effettuata attraverso la successione dei seguenti passi:

- associazione tra le aree di esposizione dell'applicazione e le famiglie di firme Fortify;
- selezione delle firme di rilevamento.

L'obiettivo primario consiste nel giungere ad una restrizione del campo di osservazione agli aspetti peculiari dell'applicazione sotto esame e nel fornire un'interpretazione delle problematiche rilevate che ne consenta una valutazione oggettiva e focalizzata sulle principali funzioni da essa utilizzate.

In quest'ottica diventa indispensabile procedere all'identificazione del piano di esposizione dell'applicazione, così che possa essere possibile, in base alle caratteristiche peculiari dell'applicazione sotto esame, definire quali aree funzionali devono essere osservate.

Prima di descrivere i passi sopra menzionati è bene presentare l'organizzazione delle firme di rilevamento all'interno del software Fortify.

5.5.1 Firme di rilevamento Fortify

Le firme di rilevamento all'interno di Fortify Audit Workbench sono suddivise in otto domini (o "kingdom"); questi sono:

- input validation and representation;
- API abuse;
- security features;
- time and state;
- error handling;
- code quality;
- encapsulation;
- environment.

Alcune di queste firme sono relative allo stile di programmazione, alcune verificano problematiche certe quali, ad esempio, alcuni costrutti di codice certamente sfruttabili, mentre altre segnalano semplicemente che vengono effettuate chiamate esterne. In breve, il rischio, come per tutti gli strumenti software di questo genere, in cui alcune forme determinate possono configurarsi come vulnerabilità soltanto in certi casi e non in maniera generale, è di eseguire un'analisi del codice ottenendo diverse migliaia di occorrenze, la maggior parte delle quali identificabili come falsi positivi oppure, escludendo troppe firme, di non rilevare vulnerabilità gravi effettivamente presenti nel codice, ottenendo falsi negativi.

Di seguito vengono descritti i kingdom utilizzati da Fortify per la suddivisione delle firme di rilevamento in famiglie. Tali Kingdom consentono anche la suddivisione delle problematiche di sicurezza in base a quanto riscontrato.

Input validation and representation

Le problematiche relative alla validazione dell'input sono tipicamente causate da metacaratteri, codifica dei caratteri e rappresentazioni di valori numerici. Tali problematiche di sicurezza derivano principalmente da una mancata verifica dell'input e includono tipicamente buffer overflow, cross-site scripting e SQL injection. Attualmente, esse sono le più diffuse e pericolose dal punto di vista della sicurezza applicativa.

API Abuse

Le API possono essere essenzialmente viste come un contratto tra un chiamante ed un chiamato. Nel momento in cui uno dei due elementi, tipicamente il chiamante, non rispetta tutti i vincoli del contratto, si aprono le porte a problematiche di sicurezza. Ad esempio, se un programma non effettua la chiamata alla funzione `chdir()` dopo aver chiamato `chroot()`, viene violato il contratto che specifica le modalità per cambiare la root directory in sicurezza.

Security features

Nonostante la sicurezza del software vada ben oltre le sole funzionalità di sicurezza, queste rivestono, comunque, un ruolo di primo piano e assolutamente non sottovalutabile. In particolar modo, con il termine "Security features", si intendono le funzionalità di autenticazione, gestione degli accessi, confidenzialità, crittografia e gestione dei privilegi. Una password di accesso ad un database inserita come costante all'interno di un codice è un tipico esempio di funzionalità di sicurezza gestita male.

Time and state

Tipicamente, si guarda ad un codice sorgente pensando che quanto scritto venga eseguito in modo ordinato, senza interruzioni. Ovviamente, sistemi operativi multitasking e CPU multicore, però, non utilizzano questo approccio. Le problematiche di questa tipologia mirano a sfruttare lo scostamento tra il modello di pensiero del programmatore e quello che avviene nel mondo reale. Esse sono causate da interazioni inattese tra thread, processi, tempo e dati.

Error handling

Benché la gestione degli errori sia essenzialmente una problematica riconducibile all'impiego di specifiche API, la frequenza con cui questi si verificano e la rilevanza che essi rivestono consigliano l'adozione di una specifica collocazione funzionale.

La problematica principale legata alla mancata o alla cattiva gestione degli errori consiste nel rivelare più informazioni di quante non sia effettivamente necessario. Tali informazioni in eccesso potrebbero essere utilizzate da un ipotetico attaccante per sferrare un attacco mirato nei confronti di un applicativo.

Code quality

Una bassa qualità del codice può sfociare in un comportamento anomalo ed imprevedibile dell'applicativo. Per un attaccante può, quindi, essere più facile sottoporre l'applicazione a situazioni di elevato stress. Un tipico esempio è rappresentato da un "infinite loop", ovvero un ciclo dal quale non è possibile uscire, evenienza che potrebbe causare un blocco del sistema. Sicurezza del codice e qualità del codice sono indissolubilmente legati.

Encapsulation

Con il termine "encapsulation" si intende la descrizione e la definizione di confini (interni ed esterni all'applicativo), nell'ambito dei quali, informazioni e dati possono e devono circolare. Il mancato contenimento di variabili, funzioni o codice può condurre alla fuga di informazioni e, in casi estremi, alla compromissione dell'intero sistema.

Environment

All'interno di questo contesto di "ambiente" vengono collocate tutte quelle problematiche che, pur non riguardando direttamente il codice sorgente, rivestono, comunque, un ruolo di elevata criticità per la sicurezza del sistema. Un esempio di problematiche legate all'environment è costituito da una cattiva configurazione di file o di flag utilizzati in sede di compilazione.

5.5.2 Associazione tra le aree di esposizione dell'applicazione e le famiglie di firme Fortify

Dopo aver individuato le aree di esposizione secondo la classificazione della Tabella 4.2, il passo operativo successivo consiste nell'individuare l'associazione tra le aree di esposizione dell'applicazione identificate e le famiglie Fortify. La Tabella 5.1 esprime tale associazione.

In un momento successivo deve essere effettuata la valutazione delle informazioni ottenute a corredo del codice sorgente posto sotto esame. Nel caso in cui venisse rilevata la non completezza delle informazioni o la necessità di integrazione delle stesse, si dovrebbe procedere alla richiesta di approfondimento ed integrazione.

Macro-Categoria	*Categoria*	*Famiglia Fortify*
Validazione dell'input	Script, Servlet e CGI	Input Validation and Representation
Bound checking e problematiche di overflow	Stack Overflow	Input Validation and Representation
	Off-by-one/Off-by-few	Input Validation and Representation
	Format String	Input Validation and Representation
	Heap Overflow	Input Validation and Representation
	Integer Overflow ed altri errori logici di programmazione	Input Validation and Representation
Session management	Session Stealing ed Hjihacking	Security Features
	Accesso ad aree non autorizzate	Security Features
Crittografia	Sniffing ed algoritmi crittografici deboli	Security Features
	Brute Forcing	Security Features
	Rainbow Table e Salt Value	Security Features
	Archiviazione insicura	Security Features
Error e time handling	User Enumeration	Error Handling
	Information Disclosure	Error Handling
	Directory Listing	Error Handling
	Denial Of Service	Input Validation and Representation, Error Handling
	Race Condition	Time and State
	Privilege Escalation e Bypassing dei permessi utente	Error Handling
Processi di tracciamento	Errori comuni nei meccanismi di tracciamento	Error Handling

Tabella 5.1. Associazione tra aree di esposizione e famiglie Fortify

5.5.3 Metodologia per la selezione delle firme di rilevamento

Per realizzare l'attività di selezione delle firme di rilevamento in ambito di code inspection devono essere valutate tutte le informazioni ottenute a corredo del codice; in particolar modo, è necessario tener conto delle informazioni disponibili relativamente a:

- linguaggio di programmazione impiegato;
- tipologia di risorse accedute;
- esposizione dell'applicazione/API;
- tipologia di attività effettuate;
- necessità di analisi specifiche.

Nel corso dell'attività, una volta individuati gli obiettivi della contestualizzazione, devono essere effettuate tutte quelle reiterazioni del processo di selezione necessarie al loro raggiungimento.

È chiaro che le operazioni di selezione delle firme di rilevamento per l'ispezione del codice sorgente dipendono, comunque, in massima parte, dall'esperienza degli analisti che le effettuano e dalla conoscenza approfondita dello strumento di analisi

impiegato. Tuttavia, procedendo per passi successivi, è possibile ottenere validi risultati con un tempo di apprendimento relativamente breve.

Ai fini della selezione delle firme più appropriate e in grado di fornire risultati maggiormente significativi e consistenti, assumono particolare interesse i criteri descritti di seguito in ordine decrescente d'importanza.

Linguaggio di programmazione impiegato

Al di là delle vulnerabilità "standard", ciascun linguaggio è caratterizzato da peculiari problematiche di sicurezza. Andranno selezionate, quindi, quelle firme le cui vulnerabilità associate possono essere ritrovate all'interno del linguaggio di programmazione impiegato e ad esso sintatticamente compatibili. Non avrebbe senso, infatti, selezionare firme che attivino il motore d'analisi associato ad un linguaggio diverso, e dunque incomprensibile, rispetto a quello impiegato.

Tipologia di risorse accedute

In base all'analisi della documentazione disponibile, o delle eventuali richieste di approfondimento inviate, dovrebbero essere abilitate tutte le regole, o categorie di regole, relative alla tipologia di risorsa interessata (ad esempio, vulnerabilità legate all'errata gestione dei permessi su file system, vulnerabilità legate all'errata gestione dei socket, ecc.).

Entrando nel merito della singola firma, ovvero scendendo ad un maggiore livello di dettaglio, dovranno essere scartate quelle firme che, pur rientrando in una categoria coerente con l'applicazione, risultano al di fuori del contesto tecnologico. Si pensi, ad esempio, ad una famiglia di firme associata a funzioni di accesso a dischi condivisi in cui compaiono funzioni legate ad NFS per un'applicazione che impiega SMB.

In modo simile si dovrà provvedere a de-selezionare quelle firme che fanno riferimento a vulnerabilità associate a particolari chiamate che risultano assenti dall'applicazione in esame. Se l'applicazione non si appoggia su una base di dati, allora tutte le firme legate alle chiamate a DBMS o a riferimenti SQL dovranno essere disattivate.

Esposizione dell'applicazione/API

Alcune vulnerabilità, come, ad esempio, SQL Injection o Cross Site Scripting, possono essere sfruttate solo ed esclusivamente in quei casi in cui un applicativo renda disponibili verso l'esterno specifici servizi (ad esempio, un form Web). L'abilitazione delle regole deve, pertanto, tener conto delle interfacce offerte dall'applicazione e delle specifiche API messe a disposizione da quest'ultima.

Tipologia di attività effettuate

In tutti i casi in cui l'applicazione si avvalga di costrutti standard, tipici di ciascun linguaggio, per l'accesso a specifiche attività, ad esempio, per la gestione dell'input/output o del sistema di log, le regole legate a tali funzioni dovrebbero essere abilitate.

Necessità di analisi specifiche

Nel caso in cui fosse richiesta un'analisi più approfondita, in particolar modo nel corso della prima fase di analisi, si dovrebbero selezionare quelle regole che, in base alle ulteriori informazioni, consentono di aggiungere ulteriori dettagli ed informazioni utili.

Può risultare utile attivare alcune firme particolari basandosi sull'esperienza di risultati ottenuti da precedenti analisi o da segnalazioni derivate da rapporti d'incidente. Infatti, alcune vulnerabilità potrebbero essere legate al modo con cui gli analisti-programmatori di un certo fornitore sono abituati a costruire parti degli algoritmi che compongono sezioni di codice di un'applicazione. Un'altra causa di vulnerabilità potrebbe essere una sezione di codice per l'accesso ad un altro sistema che contiene in sé vulnerabilità e viene distribuita a vari programmatori per essere integrata all'interno del codice da loro prodotto. In questo caso è bene mantenere attiva tale firma per assicurarsi che una vulnerabilità certa non venga reintrodotta nelle applicazioni aziendali.

6

Sviluppo in sicurezza del codice C/C++

Questo capitolo ha lo scopo di presentare le principali pratiche di sviluppo di codice C e C++ in modo sicuro. Inizieremo con il raccontare brevemente la storia di questi linguaggi in quanto la conoscenza di tale storia ci aiuta a capire il perché di alcune loro caratteristiche fondamentali. Successivamente esamineremo una serie di "best practices" per lo sviluppo di codice sicuro, riconosciute dai massimi esperti a livello internazionale. La parte finale del capitolo sarà dedicata alla descrizione delle categorie di vulnerabilità in C/C++ che è possibile riscontrare con il software Fortify.

6.1 Introduzione ai linguaggi C/C++

Il C è un linguaggio di programmazione di alto livello, seppur semanticamente correlabile al linguaggio macchina e all'assembly. Esso discende da due linguaggi, il BCPL e il B, che non prevedevano tipi di dati: ogni dato occupava una parola di memoria e la responsabilità di interpretare un dato come un numero intero piuttosto che come un numero reale ricadeva tutta sul programmatore.

Il linguaggio C si sviluppò a partire dal B grazie a Dennis Ritchie presso i Bell Laboratories e fu implementato originariamente su un computer DEC PDP-11 nel 1972. Il C riprese molti concetti utili dei precedenti linguaggi e, in più, aggiunse il concetto fondamentale di tipo di dato.

Esso nacque inizialmente come linguaggio di sviluppo del sistema operativo Unix, mentre oggi, mantenendo una certa logicità con le sue origini storiche, viene prevalentemente utilizzato per la progettazione di applicazioni in ambiente Unix.

Il primo tentativo di standardizzare le specifiche del linguaggio, che sin dai primi anni di nascita andava sfaldandosi in dialetti multipli, culminò con la definizione nel 1989 di quello che oggi è comunemente denominato ANSI C o C89. L'ultimo documento di livello internazionale per la definizione dello standard è

stato divulgato nel 1999 dalla commissione congiunta ISO/IEC e prende il nome di 9899:1999 o, più semplicemente, C99.

Il C, così standardizzato, è indipendente dall'hardware ed è disponibile nella maggior parte dei sistemi; pertanto, le applicazioni scritte in C possono essere spesso eseguite, con qualche (o, addirittura, senza alcuna) modifica, su gran parte dei sistemi esistenti.

Il C++ è, invece, un linguaggio di programmazione orientato agli oggetti ideato nel 1983 da Bjarne Stroustrup come evoluzione naturale del C presso i Bell Laboratories. Gli oggetti sono essenzialmente componenti software riutilizzabili, modellati sul mondo reale.

Una progettazione orientata agli oggetti può rendere i gruppi di programmazione molto più produttivi rispetto ai vecchi metodi. I programmi orientati agli oggetti sono più semplici da leggere, da correggere e da modificare.

Il C++ è molto utilizzato per lo sviluppo di applicazioni nel settore delle telecomunicazioni, per la progettazione di software real-time e per il controllo dei processi industriali.

Il primo tentativo di standardizzare il linguaggio risale al 1998 con la definizione della ISO/IEC 14882, oggi resa obsoleta dalla revisione effettuata nel 2003 (ISO/IEC 14882:2003).

Il C++ è un linguaggio ibrido: è possibile programmare in C++ nel vecchio stile C, nel nuovo stile orientato agli oggetti, o in entrambi.

6.2 Best practices considerate per l'attività di code inspection

Nel seguito viene illustrata una serie di "best practices" per lo sviluppo di codice sicuro, riconosciute dai massimi esperti a livello internazionale. Tali direttive sono conformi con i dettami degli standard *CERT C/C++ Programming Language Secure Coding*. Chiaramente queste direttive hanno rappresentato le principali linee guida per l'attività di code inspection con i linguaggi C/C++ da noi effettuata nell'ambito della presente tesi.

6.2.1 Dichiarazioni

Per quanto concerne le dichiarazioni:

- È consigliato dimensionare gli array non utilizzando costanti numeriche, ma piuttosto costanti simboliche definite. La Tabella 6.1 mette in contrapposizione una forma scorretta con una corretta.
- È consigliato dichiarare le costanti utilizzando la parola chiave `const`. La Tabella 6.2 evidenzia come andrebbe dichiarata una costante.

Forma Scorretta	*Forma Corretta*
`int mesi[13];`	`int mesi[TOT_MESI + 1];`

Tabella 6.1. C/C++: Dimensionamento di un array

Forma Scorretta	*Forma Corretta*
`int mesi = 12;`	`const unsigned int mesi = 12;`

Tabella 6.2. C/C++: Costanti dichiarate tramite la parola chiave `const`

- È consigliato dichiarare le variabili che possono avere valori positivi utilizzando la keyword `unsigned`.
- Il tipo `char` deve essere `unsigned`.
- È consigliato non utilizzare `float` e `double` quando non è necessario (calcoli scientifici).
- Occorre assicurarsi che le classi che hanno funzioni virtuali abbiano anche distruttori virtuali.

6.2.2 Inizializzazioni

Per quanto riguarda le inizializzazioni:

- Tutte le variabili locali devono essere inizializzate prima di essere utilizzate.
- Tutte le variabili locali che sono inizializzate con valori "dummy" o momentanei devono essere reinizializzate con i valori reali al momento dell'uso.
- Tutte le variabili legate ai cicli devono essere reinizializzate con l'entrata in una nuova iterazione.
- Tutte le variabili legate ai cicli devono essere reinizializzate prima di essere riutilizzate in un nuovo ciclo.
- Tutte le strutture devono essere azzerate prima del loro utilizzo.
- Tutti i buffer devono essere azzerati prima del loro utilizzo o riutilizzo.

6.2.3 Utilizzo dei tipi di dati

Stringhe

- Tutte le stringhe devono essere terminate dal carattere `NULL`. È necessario evitare errori logici di programmazione che agevolino l'insorgere di una con-

dizione contraria. È necessario riporre attenzione nell'utilizzo di funzioni che non aggiungono al termine di una stringa copiata in un buffer di destinazione il carattere `NULL` se questo non risiede nel buffer sorgente. Nella Tabella 6.3 segue un esempio di quanto appena scritto.

Forma Scorretta	*Forma Corretta*
`strncpy(dest, source, sizeof(dest));`	`cstrncpy(dest, source, sizeof(dest);` `dest[sizeof(dest) - 1] = '\0';`

Tabella 6.3. C/C++: Stringhe e carattere `NULL`

- Il codice non deve tentare di operare su una stringa (o un array di caratteri) che non è terminato dal carattere `NULL`.
- L'input proveniente dall'utente deve sempre essere convalidato e scremato da caratteri non validi (“;” “|” “!” “&” “~” “,” “"” “-” “*” “%” “'” “\” “/” “<” “>” “?” “$” “@” “:” “(” “)” “[” “]” “{” “}” “.”) prima di essere passato alle successive elaborazioni dell'applicazione (ad esempio, alla funzione `system()`).
- Occorre utilizzare le funzioni `strspn()`, `strcspn()` e `strpbrk()` per filtrare l'input utente.

Buffer

- Tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati.
- Occorre evitare l'utilizzo di funzioni che non consentono di specificare la dimensione delle stringhe copiate da un buffer sorgente ad uno di destinazione. Le funzioni considerate critiche in questo contesto e che non devono mai essere utilizzate sono: `strcpy()`, `wcscpy()`, `sprintf()`, `strcat()`, `gets()`, `scanf()`, `vsprintf()` e `wcscat()`.
- Quando i dati vengono copiati all'interno di un buffer deve essere sempre verificata la loro dimensione con quella del buffer di destinazione. Le funzioni considerate critiche per errori di bound-checking, pur avendo la possibilità di specificare la lunghezza delle stringhe soggette a copia da un buffer all'altro, sono: `strncpy()`, `wcsncpy()`, `snprintf()`, `strncat()`, `vsnprintf()`, `wcsncat()`, `mbstowcs()`, `mbsrtowcs()`, `memcpy()`, `memmove()`, `memset()`, `strxfrm()`, `vswprintf()`, `wmemset()`, `wmemcpy()`, `wmemmove()`, `wcstombs()`, `wcsrtombs()`, `wcsxfrm()` e `swprintf()`.
  Nella Tabella 6.4 sono riportati alcuni esempi di funzioni solitamente considerate sicure, ma utilizzate in modo errato.

<table>
<tr><th>Forma Scorretta</th><th>Forma Corretta</th></tr>
<tr><td><pre>char dest[512];
char *source;
// puntatore char source
// manipolabile dall'utente
strncpy(dest, source, strlen(source));</pre></td><td><pre>char dest[512];
strncpy(dest, source, sizeof(dest));
// inserimento di NULL
// alla fine di dest
...</pre></td></tr>
<tr><td><pre>#define LEN 5000
// LEN superiore alla capacità di
// contenimento massima di dest
char dest[1024];
// variabile source
// manipolabile dall'utente
char source[LEN];
memcpy(dest, source, LEN);</pre></td><td><pre>#define LEN 1024;
char dest[1024];
// variabile source
// manipolabile dall'utente
char source[LEN];
memcpy(dest, source, LEN - 1);
// inserimento di NULL
// alla fine di dest
...</pre></td></tr>
</table>

Tabella 6.4. C/C++: Buffer e problematiche di bound-checking

- Il formato delle stringhe deve sempre essere specificato nei parametri delle funzioni che lo richiedono. In questo contesto le funzioni considerate critiche e soggette a problematiche di format string overflow, se non correttamente utilizzate, sono: `printf()`, `fprintf()`, `sprintf()`, `snprintf()`, `vprintf()`, `vfprintf()`, `vsprintf()`, `vscanf()`, `vsscanf()`, `vsnprintf()`, `fscanf()`, `sscanf()`, `vwprintf()`, `vfwprintf()`, `vswprintf()`, `vfscanf()`, `wprintf()`, `fwprintf()`, `swprintf()` e `scanf()`.
  La Tabella 6.5 mostra come evitare questo tipo di problematiche.

Bitfield

- Se nel codice vengono svolte operazioni di shifting o si utilizzano bitfield, è necessario indicare le piattaforme con cui il codice è compatibile per mitigare problemi/errori di porting.

6.2.4 Macro

- Se le macro sono espanse i parametri passati non devono causare effetti collaterali. Pertanto, gli argomenti delle macro devono essere accuratamente racchiusi tre parentesi. La Tabella 6.6 riporta un esempio esplicativo.

Forma Scorretta	*Forma Corretta*
`printf(buffer1);`	`printf("%s\r\n", buffer1);`
`snprintf(dest, sizeof(dest), buf);`	`snprintf(dest, sizeof(dest), "%s", buf);`
`fprintf(FILE, num, stringa);`	`fprintf(FILE, "%d: %s%\n", num, stringa);`

Tabella 6.5. C/C++: Buffer e problematiche di format string overflow

Forma Scorretta	*Forma Corretta*

`#define max(a, b) (a) > (b) ? (a) : (b)`
`risultato = max(i, j) + 3;`
`// tutto questo viene espanso in`
`// risultato = (i) > (j) ? (i) : (j)+3;`
	`#define max(a,b) ((a) > (b) ? (a) : (b))`

Tabella 6.6. C/C++: Macro ed effetti collaterali

6.2.5 L'operatore `sizeof` ed il passaggio di dati come parametri

- Il passaggio della dimensione di una struttura dati come parametro ad una funzione deve essere effettuato in maniera corretta, tramite l'utilizzo della funzione `sizeof`. A tale proposito è obbligatorio prendere visione degli errori menzionati in Tabella 6.7 e non ripeterli.

6.2.6 Allocazione dinamica

- Lo spazio di memoria allocato dinamicamente (ad esempio, con le funzioni `malloc()`, `calloc()` e `realloc()`) deve essere appropriato rispetto alla dimensione dei dati che deve contenere.
- L'applicazione deve provvedere all'allocazione ed alla deallocazione della sua memoria. Nell'ambito della programmazione multithreaded vale lo stesso principio; ogni thread deve allocare e deallocare la propria memoria senza delegare la deallocazione ad altri thread.
- Non utilizzare, in un sorgente C++, le funzioni standard del C. A tale proposito è meglio utilizzare `new` invece di `malloc()`, `calloc()` e `realloc()`.

Forma Scorretta	*Forma Corretta*
`strlen(struttura)`	`sizeof(struttura)`
`sizeof(ptr)`	`sizeof(*ptr)`
`sizeof(*array)`	`sizeof(array)`
`// Dimensione di un solo elemento` `sizeof(array)`	`// Dimensione di un solo elemento` `sizeof(array[0])`

Tabella 6.7. C/C++: Operatore `sizeof()`

6.2.7 Deallocazione

- Gli array non devono essere cancellati come dati scalari. La Tabella 6.8 evidenzia come dovrebbe essere cancellato un array.

Forma Scorretta	*Forma Corretta*
`delete mioarray;`	`delete [ ] mioarray;`

Tabella 6.8. C/C++: Cancellazione di array

- Non devono esistere puntatori a risorse distrutte; contestualmente alla distruzione delle risorse è necessario de-referenziare i corrispettivi puntatori.
- I puntatori relativi alla memoria allocata dinamicamente devono essere impostati a `NULL` subito dopo essere stati rilasciati.
- I puntatori ottenuti tramite `malloc()`, `calloc()`, `realloc()` devono essere distrutti con `free()`; non si deve mai usare `delete`.
- I puntatori ottenuti tramite `new` devono essere distrutti con `delete` (mai usare `free()`).

- Non bisogna mai liberare un'area di memoria (ad esempio, con `free()`) già deallocata. È necessario evitare errori logici nel codice che consentano l'insorgere di problematiche di questo tipo.
- Non bisogna mai tentare di scrivere in un buffer residente in heap memory dopo la sua deallocazione. Evitare l'insorgere di errori logici di questo tipo.

6.2.8 Puntatori

- Occorre gestire opportunamente i puntatori a `NULL`; la Tabella 6.9 confronta una gestione scorretta con una corretta.

Forma Scorretta	*Forma Corretta*

`char tmpchar1 (char *s)`
`{`
` return *s;`
`}`
`//"s" == NULL -> CRASH`
	`char tmpchar1 (char *s)`
	`{`
	` if (s == NULL) return '\0';`
	` return *s;`
	`}`

Tabella 6.9. C/C++: Gestione di puntatori a `NULL`

6.2.9 Casting e problematiche di gestione delle variabili numeriche

- Il tipo `NULL` deve essere corretto mediante casting quando viene passato come parametro ad una funzione.
- È necessario ridurre al minimo le comparazioni fra interi di tipo `signed`. Se due interi di tipo `signed` vengono comparati deve essere previsto il caso "minore di zero" (`< 0`) soprattutto quando la comparazione avviene con un valore costante. La Tabella 6.10 mostra una comparazione signed/unsigned tra interi.

- Si deve evitare di utilizzare variabili `signed integer` come length specifier, ovvero come indicatori dell'allocazione/dimensione di un buffer o di un array.
- Si deve evitare che un intero, a seguito di un'operazione di moltiplicazione, addizione o sottrazione, cresca oltre il suo valore massimo o decresca sotto il suo valore minimo. Ad esempio, su architettura a 32 bit, se un intero signed a 16 bit del valore di 32767 viene incrementato di una unità, il suo valore diverrà -32768. È bene assicurarsi che questo genere di condizioni non si verifichi in alcun caso, soprattutto su input fornito dall'utente, in prossimità dell'allocazione di un buffer o della copia di dati da un buffer all'altro.

Comparazione non Signed	*Comparazione Signed*

`if ((int)val1 < (unsigned int)val2)`
`/* in questo caso unsigned ha la`
`precedenza essendo un tipo più grande`
`di signed. Entrambi i valori`
`(val1 e val2) vengono quindi convertiti`
`ad unsigned prima di essere comparati`
`*/`
	`if ((int)val < 256)`

`if ((int)val < sizeof(costante))`
`// l'operatore sizeof è unsigned`
	`if ((unsigned short)val1 < (short)val2)`
	`/* la seguente comparazione dovrebbe,`
	`in base al tipo di compilatore, essere`
	`signed perché entrambi gli short`
	`dovrebbero essere convertiti a signed`
	`integer prima di essere comparati`
	`*/`

Tabella 6.10. C/C++: Comparazione signed/unsigned tra interi

- La conversione fra interi di dimensioni diverse deve essere il più possibile evitata. La conversione di un intero di grandi dimensioni ad uno più piccolo (da 32 a 16 bit o da 16 a 8 bit) può causare il troncamento del valore memorizzato in una variabile o determinarne il cambio di segno. Anche la conversione di un intero di piccole dimensioni ad uno più grande può introdurre delle problematiche; infatti, ad esempio, la conversione dell'intero signed a 16 bit -1 in intero unsigned a 32 bit darà come risultato il valore 4.294.967.295. In particolare, sono negate tutte le conversioni riportate nella Tabella 6.11.
- Il codice non deve affidarsi a conversioni implicite e/o dedotte dal compilatore.

6.2.10 Condizioni

- I dati devono essere appropriatamente confrontati con altri dello stesso tipo, specialmente per i tipi `float` e `double`. Ad esempio, la seguente condizione:

```
if ( variabile == 0.1 )
```

  potrebbe non rivelarsi mai vera per le proprietà di arrotondamento del compilatore.
- Le variabili dichiarate come `unsigned` non devono mai essere confrontate con lo zero utilizzando l'operatore "maggiore di". Infatti la condizione:

```
if ( variabile > 0)
```

Da	*A*
`16 bit signed`	`32 bit unsigned`
`32 bit signed`	`16 bit unsigned`
`32 bit unsigned`	`16 bit signed`
`32 bit signed`	`16 bit signed`

Tabella 6.11. C/C++: Conversione fra interi da evitare

risulta sempre vera se `variabile` è `unsigned`.

- Le variabili dichiarate come `signed` non devono mai essere confrontate con `TRUE`, come nel seguente caso:

```
if (variabile)
```

  Se, ad esempio, `variabile` può assumere un valore negativo è meglio prevedere questo caso con un controllo del tipo:

```
if (variabile != 0)
```

  oppure, in un modo ancora più esplicito, controllando il segno dell'intero.

6.2.11 Controllo del flusso

Variabili di controllo

- È obbligatorio utilizzare sempre un limite superiore inclusivo e un limite inferiore esclusivo, come nell'esempio seguente:

Istruzioni `Switch`

- Ogni blocco di codice corrispondente ad un `case` di uno `switch` deve essere terminato dalla parola chiave `break`.
- Ogni `switch` deve avere un caso di `default`.

Forma Scorretta	*Forma Corretta*
`x >= 23 e x <= 42`	`x >= 23 e x < 43`

Tabella 6.12. C/C++: Variabili di controllo e loro limiti

6.2.12 Passaggio di argomenti

- I tipi di dati esterni non devono essere passati "per valore" (by value).
- I vettori e le strutture devono sempre essere passati per indirizzo o per riferimento.
- È auspicabile utilizzare la parola chiave `const` per i paramatri (strutture o vettori) passati in ingresso ad una funzione.

6.2.13 Valori di ritorno

- I tipi di dati devono essere appropriati per memorizzare i valori di ritorno delle funzioni.
- Se i parametri delle funzioni sono passati come riferimenti `const`, i valori di ritorno devono anche essere ritornati come riferimenti `const`.

6.2.14 Chiamate a funzioni

- Il file pointer passato come argomento ad una chiamata a `fprintf()` deve essere inizializzato.
- Ogni chiamata di funzione deve avere i parametri corretti, coerenti con il tipo ed il formato del corrispettivo prototipo.

6.2.15 File

- Ogni nome di file temporaneo deve essere unico e non predicibile.
- Ogni puntatore a file deve essere chiuso prima di essere riutilizzato.

6.2.16 Gestione degli errori

- I valori di ritorno di tutte le chiamate di sistema devono essere controllati per determinare lo stato di esecuzione del programma. Funzioni come `perror()`, `ferror()` ed `strerror()` e la costante `errno` devono essere utilizzate per determinare o riportare all'utente il tipo di errore occorso.

- `errno` non deve essere dichiarato manualmente come un `extern` se risiede in uno degli `include` dell'implementazione C/C++ utilizzata.
- Al verificarsi di un errore critico o imprevisto, a seguito di una chiamata di sistema, tutti i puntatori e le aree di memoria utilizzate devono essere dereferenziati/disallocati prima della chiusura del programma.

6.2.17 Sicurezza dell'applicazione

- I risultati dei controlli, delle procedure di sicurezza ed i relativi dati non devono risiedere in memoria per lunghi periodi. Ad esempio, le chiavi crittografiche devono permanere in memoria solo per il tempo necessario al loro utilizzo e devono essere sovrascritte con dati casuali o garbage al termine del loro impiego.
- I dati critici non devono mai essere serializzati.

6.3 Vulnerabilità per i linguaggi di programmazione C/C++ riscontrabili tramite Fortify

Come detto in precedenza, per la nostra attività di code inspection abbiamo utilizzato il software Fortify Source Code Analyzer.

Tale software, per quanto riguarda i linguaggi di programmazione C/C++, è in grado di riscontrare una serie di vulnerabilità, ciascuna legata a un determinato dominio.

Più specificatamente, le vulnerabilità che è possibile riscontrare con il software Fortify Source Code Analyzer sono le seguenti:

- *Access Control: Database*. Questa vulnerabilità riguarda il fatto che, senza un opportuno controllo degli accessi, l'esecuzione di un comando SQL contenente una chiave primaria controllata dall'utente può consentire ad un attaccante di visualizzare record senza autorizzazione. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Buffer Overflow*. Questa vulnerabilità riguarda il fatto che la scrittura fuori dai limiti di un blocco di memoria allocata può causare la corruzione dei dati, il blocco del programma o l'esecuzione di codice malevolo. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Code Correctness: Arithmetic Operation on Boolean*. Questa vulnerabilità riguarda il fatto che il programma impiega un operatore aritmetico su un valore booleano, il che potrebbe non realizzare ciò che il programmatore ha in mente. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Erroneous Synchronization*. Questa vulnerabilità riguarda il fatto che, se un thread fallisce lo sblocco di un mutex dopo aver segnalato

altri thread, questi ultimi rimarranno bloccati in attesa del mutex. Il dominio di riferimento per questa vulnerabilità è "Time and State".

- *Code Correctness: Function Not Invoked.* Questa vulnerabilità riguarda il fatto che, a causa della mancanza di parentesi di chiusura, un'espressione si riferisce al valore del puntatore alla funzione piuttosto che al valore di ritorno della stessa. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Function Returns Stack Address.* Questa vulnerabilità riguarda il fatto che la restituzione dell'indirizzo di uno stack variabile può causare comportamenti non desiderati del programma, tipicamente sotto forma di blocco. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Macro Misuse.* Questa vulnerabilità riguarda il fatto che famiglie di funzioni che operano su risorse condivise e che sono implementate come macro su alcune piattaforme devono essere chiamate nello stesso ambito del programma. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Premature Thread Termination.* Questa vulnerabilità riguarda il fatto che, se un processo genitore termina l'esecuzione normalmente prima dei thread che ha generato, i thread potrebbero terminare prematuramente. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Command Injection (Data Flow).* Questa vulnerabilità riguarda il fatto che l'esecuzione di comandi che includono input utente non validato può far si che un'applicazione agisca a beneficio di un attaccante. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Command Injection (Semantic).* Questa vulnerabilità riguarda il fatto che l'esecuzione di comandi senza la specificazione del percorso assoluto può consentire ad un attaccante di eseguire codice malevolo modificando il `$PATH` o altri aspetti dell'ambiente di esecuzione del programma. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Dangerous Function.* Questa vulnerabilità riguarda il fatto che funzioni che non possono essere utilizzate in sicurezza non dovrebbero mai essere previste. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Dead Code.* Questa vulnerabilità riguarda il fatto che l'istruzione in questione non sarà mai eseguita. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Denial of Service.* Questa vulnerabilità riguarda il fatto che un attaccante potrebbe causare il blocco di un programma oppure renderlo non disponibile agli utenti autorizzati. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Directory Restriction.* Questa vulnerabilità riguarda il fatto che un utilizzo non corretto della chiamata di sistema `chroot()` potrebbe consentire ad un attac-

cante di aggirare il corrispettivo ambiente protetto. Il dominio di riferimento per questa vulnerabilità è “API Abuse”.

- *Double Free*. Questa vulnerabilità riguarda il fatto che una doppia chiamata `free()` sullo stesso indirizzo di memoria può causare un buffer overflow. Il dominio di riferimento per questa vulnerabilità è “Code Quality”.
- *Format String*. Questa vulnerabilità riguarda il fatto che un mancato controllo della formattazione delle stringhe da parte delle funzioni può causare un buffer overflow. Il dominio di riferimento per questa vulnerabilità è “Input Validation and Representation”.
- *Heap Inspection*. Questa vulnerabilità riguarda il fatto che non si dovrebbe utilizzare `realloc()` per ridimensionare i buffer contenenti informazioni sensibili. Il dominio di riferimento per questa vulnerabilità è “API Abuse”.
- *Heap Inspection: Swappable Memory*. Questa vulnerabilità riguarda il fatto che non si dovrebbe utilizzare `VirtualLock()` per bloccare le pagine che contengono informazioni sensibili in memoria. Il dominio di riferimento per questa vulnerabilità è “API Abuse”.
- *Illegal Pointer Value*. Questa vulnerabilità riguarda il fatto che una funzione potrebbe restituire un puntatore a un indirizzo fuori dal buffer cercato. Le successive operazioni sul puntatore possono avere conseguenze non desiderate. Il dominio di riferimento per questa vulnerabilità è “Input Validation and Representation”.
- *Insecure Compiler Optimization*. Questa vulnerabilità riguarda il fatto che le operazioni non corrette di cancellazione di dati sensibili dalla memoria potrebbero compromettere la sicurezza. Il dominio di riferimento per questa vulnerabilità è “Environment ”.
- *Insecure Randomness*. Questa vulnerabilità riguarda il fatto che l’impiego di generatori standard di numeri pseudo-casuali potrebbe non contrastare sufficientemente attacchi crittografici. Il dominio di riferimento per questa vulnerabilità è “Security Features”.
- *Insecure Temporary File*. Questa vulnerabilità riguarda il fatto che la creazione e l’utilizzo scorretto dei file temporanei può rendere vulnerabili i dati dell’applicazione e del sistema. Il dominio di riferimento per questa vulnerabilità è “Time and State”.
- *Integer Overflow*. Questa vulnerabilità riguarda il fatto che, non tenere in conto gli integer oveflow potrebbe provocare errori logici o buffer overflow. Il dominio di riferimento per questa vulnerabilità è “Input Validation and Representation”.
- *Least Privilege Violation*. Questa vulnerabilità riguarda il fatto che l’elevato livello di privilegi richiesto per eseguire operazioni come `chroot()` dovrebbe essere abbassato appena è terminata l’esecuzione di questa operazione. Il dominio di riferimento per questa vulnerabilità è “Security Features”.

- *Log Forging.* Questa vulnerabilità riguarda il fatto che la scrittura dei file di tracciamento da parte di input utente non validato potrebbe consentire ad un attaccante di falsificare i dati di accesso al sistema o di inserire contenuto malevolo nei dati di tracciamento. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Memory Leak.* Questa vulnerabilità riguarda il fatto che la memoria è allocata, ma mai liberata. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Memory Leak: Reallocation.* Questa vulnerabilità riguarda il fatto che il programma ridimensiona un blocco di memoria allocata. Se il ridimensionamento fallisce il blocco originario è visualizzabile. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Missing Check against Null.* Questa vulnerabilità riguarda il fatto che è sempre necessario verificare la corretta gestione delle funzioni che potrebbero restituire NULL come valore di ritorno ad una chiamata. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Null Dereference.* Questa vulnerabilità riguarda il fatto che la gestione scorretta dei puntatori a NULL potrebbe causare un "segmentation fault". Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Obsolete.* Questa vulnerabilità riguarda il fatto che l'utilizzo di funzioni deprecate o antiquate potrebbe evidenziare un cattivo codice. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Often Misused: Authentication (getlogin).* Questa vulnerabilità riguarda il fatto che la funzione `getlogin()` può essere facilmente ingannata. Non basare le successive elaborazioni sullo username restituito. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Authentication (gethostbyname — gethostbyaddr).* Questa vulnerabilità riguarda il fatto che un attaccante potrebbe aver ottenuto il controllo di un server DNS. Non validare intrinsecamente le informazioni ottenute. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Exception Handling (_alloc).* Questa vulnerabilità riguarda il fatto che la funzione `_alloc()` potrebbe causare uno stack overflow. Il suo utilizzo dovrebbe essere ridotto ai casi strettamente indispensabili. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Exception Handling (EnterCriticalSection — InitializeCriticalSection).* Questa vulnerabilità riguarda il fatto che `EnterCriticalSection()` potrebbe causare un'eccezione, esponendo l'applicazione ad un potenziale blocco. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: File System (getwd).* Questa vulnerabilità riguarda il fatto che chiamare `getwd()` con un buffer di dimensioni non adeguate potrebbe condurre ad un buffer overflow. Il dominio di riferimento per questa vulnerabilità è "API Abuse".

- *Often Misused: File System (readlink)*. Questa vulnerabilità riguarda il fatto che la funzione `readlink()` non termina con un NULL. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: File System (realpath)*. Questa vulnerabilità riguarda il fatto che il buffer passato a `realpath()` dovrebbe contenere almeno `PATH_MAX` byte. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: File System (umask)*. Questa vulnerabilità riguarda il fatto che spesso la maschera utilizzata per la funzione `umask()` viene confusa con quella della funzione `chmod()`. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: File System (Windows)*. Questa vulnerabilità riguarda il fatto che un buffer di grandezza inaspettata da parte di una funzione di manipolazione del percorso potrebbe causare un buffer overflow. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Privilege Management*. Questa vulnerabilità riguarda il fatto che non adeguarsi al principio di "least privilege" aumenta il rischio di esporsi ad ulteriori problematiche di sicurezza. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Strings*. Questa vulnerabilità riguarda il fatto che funzioni che convertono tra stringhe Multibyte e Unicode sono particolarmente soggette a buffer overflow. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Often Misused: Strings (_mbs*)*. Questa vulnerabilità riguarda il fatto che la famiglia di funzioni `_mbs` è suscettibile di buffer overflow quando si utilizzano stringhe multibyte malformate. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Password Management*. Questa vulnerabilità riguarda il fatto che il salvataggio delle password in testo chiaro potrebbe compromettere l'intero sistema. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Password Management: Hardcoded Password*. Questa vulnerabilità riguarda il fatto che le password hardcoded possono compromettere seriamente la sicurezza del sistema. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Password Management: Weak Cryptography*. Questa vulnerabilità riguarda il fatto che l'offuscamento di una password con una codifica debole non protegge la password stessa. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Path Manipulation*. Questa vulnerabilità riguarda il fatto che, consentire all'input utente il controllo dei percorsi usati nelle operazioni sul file system potrebbe permettere ad un attaccante di accedere o modificare le risorse protette di sistema. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".

- *Poor Style: Redundant Initialization.* Questa vulnerabilità riguarda il fatto che l'assegnazione di un valore ad una variabile che non viene mai utilizzata è uno spreco. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Value Never Read.* Questa vulnerabilità riguarda il fatto che è uno spreco assegnare un valore ad una variabile che non viene mai letta. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Variable Never Used.* Questa vulnerabilità riguarda il fatto che la variabile non è mai utilizzata. Il dominio di riferimento per questa vulnerabilità è "Code Quality ".
- *Portability Flaw.* Questa vulnerabilità riguarda il fatto che le funzioni con implementazioni inconsistenti tra diversi sistemi operativi e tra differenti versioni di sistemi operativi causano problemi di portabilità. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Privacy Violation.* Questa vulnerabilità riguarda il fatto che il trattamento non corretto di informazioni private, come le password dei clienti o i numeri di sicurezza sociale, potrebbe compromettere la privacy degli utenti ed è spesso illegale. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Process Control (Data Flow).* Questa vulnerabilità riguarda il fatto che, consentire il caricamento di librerie esterne da una fonte non validata potrebbe permettere l'esecuzione di codice arbitrario. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Process Control (Semantic).* Questa vulnerabilità riguarda il fatto che, consentire il caricamento di librerie senza la specificazione di un percorso assoluto potrebbe permetterne la loro sostituzione con librerie arbitrarie da parte di un attaccante. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Race Condition: File System Access.* Questa vulnerabilità riguarda il fatto che la finestra temporale tra il controllo delle proprietà del file ed il suo utilizzo potrebbe essere sfruttata per lanciare un attacco di privilege escalation. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *Race Condition: Signal Handling.* Questa vulnerabilità riguarda il fatto che l'installazione dello stesso handler di segnali per tanti segnali potrebbe condurre ad una "race condition" quando diversi segnali lo raggiungono in istanti ravvicinati. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *Resource Injection.* Questa vulnerabilità riguarda il fatto che, consentire all'input utente di controllare gli identificatori delle risorse potrebbe permettere ad un attaccante l'accesso o la modifica di risorse protette di sistema. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Setting Manipulation.* Questa vulnerabilità riguarda il fatto che, consentire la modifica dei parametri di funzionamento potrebbe causare un blocco del

servizio o comportamenti imprevisti dell'applicativo. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".

- *SQL Injection*. Questa vulnerabilità riguarda il fatto che, consentire ad un utente la costruzione di un'espressione dinamica SQL potrebbe permettere ad un attaccante di modificare opportunamente l'espressione per eseguire comandi arbitrari. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *String Termination Error*. Questa vulnerabilità riguarda il fatto che, fare affidamento sui terminatori delle stringhe potrebbe portare a un buffer overflow. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *System Information Leak*. Questa vulnerabilità riguarda il fatto che, rivelare dati di sistema o informazioni di debug può favorire i piani di attacco di un malintenzionato. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Type Mismatch: Negative to Unsigned*. Questa vulnerabilità riguarda il fatto che la funzione dovrebbe restituire un valore `unsigned` ma in certi casi restituisce un valore negativo. Il dominio di riferimento per questa vulnerabilità è "Code Quality ".
- *Type Mismatch: Signed to Unsigned*. Questa vulnerabilità riguarda il fatto che la funzione dovrebbe restituire un valore `unsigned` ma in certi casi restituisce un valore `signed`. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unchecked Return Value*. Questa vulnerabilità riguarda il fatto che, la mancata considerazione di un valore restituito da un metodo potrebbe portare il programma a lasciarsi sfuggire stati e condizioni imprevisti. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Undefined Behavior*. Questa vulnerabilità riguarda il fatto che, il comportamento di una funzione non è definito a meno che il suo parametro di controllo sia impostato su un valore specifico. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Uninitialized Variable*. Questa vulnerabilità riguarda il fatto che il programma potrebbe utilizzare una variabile prima che venga inizializzata. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unreleased Resource*. Questa vulnerabilità riguarda il fatto che il programma potrebbe fallire il rilascio di risorse di sistema. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unreleased Resource: Synchronization*. Questa vulnerabilità riguarda il fatto che il programma fallisce il rilascio di un blocco che esso detiene; ciò potrebbe condurre ad un deadlock. Il dominio di riferimento per questa vulnerabilità è "Code Quality".

- *Unsafe Reflection*. Questa vulnerabilità riguarda il fatto che un attaccante potrebbe essere in grado di modificare il percorso del flusso di controllo dell'applicazione aggirando i controlli di sicurezza. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Use After Free*. Questa vulnerabilità riguarda il fatto che, fare riferimento ad un indirizzo di memoria dopo che questa è stata liberata potrebbe causare il blocco del programma. Il dominio di riferimento per questa vulnerabilità è "Code Quality".

La conoscenza di vulnerabilità è cruciale per poter procedere a ricercare le stesse nell'ambito del progetto connesso con la presente tesi. Di questa ricerca parleremo nel dettaglio nel Capitolo 8.

7

Sviluppo in sicurezza del codice Java

Questo capitolo intende presentare le principali pratiche di sviluppo in sicurezza di codice Java. Inizieremo con il raccontare brevemente la storia di questo linguaggio, evidenziandone le caratteristiche peculiari che lo rendono uno dei linguaggi di programmazione attualmente più utilizzati; tali caratteristiche, infatti, giocano un ruolo importante nelle politiche di sicurezza relative a questo linguaggio. Successivamente esamineremo una serie di "best practices" per lo sviluppo di codice sicuro Java prendendo in considerazione anche le relative Applet e le relative Servlet. Nella parte finale del capitolo esporremo le categorie di vulnerabilità in Java che è possibile riscontrare con il software Fortify.

7.1 Introduzione al linguaggio Java

Java è un linguaggio di programmazione orientato agli oggetti, derivato dal C. Le sue origini risalgono al 1991 quando un gruppo di Sun Microsystem, guidato da James Gosling e Patrick Naughton, progettò un linguaggio per l'utilizzo di elettrodomestici; in quest'ambito, però, non ottenne i consensi sperati.

I successi incominciarono a vedersi solo quattro anni dopo quando le caratteristiche peculiari del linguaggio incominciarono ad emergere; Java fu annunciato ufficialmente il 23 maggio 1995 durante l'evento SunWorld.

La piattaforma di programmazione Java è fondata sul linguaggio stesso, sulla Java Virtual Machine (JVM) e sulle API.

La caratteristica principale del linguaggio, cioè l'orientamento agli oggetti, si riferisce a un moderno metodo di programmazione e progettazione la cui idea di base è la rappresentazione, nella progettazione del software, delle entità reali o astratte che compongono il problema sotto forma di oggetti. Questi ultimi sono caratterizzati da proprietà e metodi applicabili sugli oggetti stessi, che possono, ad esempio, modificarne lo stato o estrarne informazioni. I programmi scritti in Java

possono essere unicamente orientati agli oggetti e, di conseguenza, tutto il codice deve essere necessariamente incluso in una classe. Sebbene Java possa operare sia su oggetti che su tipi di dati primitivi, è considerato un linguaggio ad oggetti "puro", ovvero nel quale gli oggetti sono le entità di base del linguaggio, anziché essere costruiti partendo da costrutti ad un livello di astrazione inferiore.

La neutralità rispetto all'architettura hardware/software del linguaggio ne implica la portabilità; questo significa che l'esecuzione di programmi scritti in Java ha un comportamento simile su piattaforme diverse. Si dovrebbe essere in grado di scrivere il programma una volta e farlo eseguire ovunque secondo il paradigma, appunto, "Scrivi una volta, esegui ovunque". Ciò è possibile tramite la compilazione del codice di Java in un linguaggio intermedio chiamato "bytecode", basato su istruzioni semplificate che ricalcano il linguaggio macchina. Il bytecode verrà, quindi, eseguito da una macchina virtuale.

Le prime implementazioni del linguaggio usavano una macchina virtuale che interpretava il bytecode per ottenere la massima portabilità. Questa soluzione, però, si rilevò poco efficiente, in quanto i programmi interpretati erano molto lenti in fase di esecuzione. Per tale motivo tutte le implementazioni recenti di macchine virtuali Java hanno incorporato un compilatore "Just In Time" (JIT compiler), ovvero un compilatore interno che, al momento del lancio, traduce al volo il programma bytecode Java in un normale programma nel linguaggio macchina del computer ospite. Inoltre, questa ricompilazione è dinamica, ovvero la virtual machine analizza costantemente il modello di esecuzione del codice (profiling) e ottimizza ulteriormente le parti più frequentemente eseguite mentre il programma è in esecuzione. Questi accorgimenti, a prezzo di una piccola attesa in fase di lancio del programma, permettono di avere delle applicazioni Java decisamente più veloci e leggere. Tuttavia, anche così, Java resta un linguaggio meno efficiente dei linguaggi compilati, come il C++, scontando il fatto di possedere degli strati di astrazione in più e di implementare una serie di automatismi, come il garbage collector che, se da un lato fanno risparmiare tempo ed errori in fase di sviluppo dei programmi, dall'altro consumano memoria e tempo di CPU in fase di esecuzione degli stessi.

Java è stato, inoltre, uno dei primi sistemi a fornire un largo supporto per l'esecuzione del codice da sorgenti remote. Un'applet Java è una particolare applicazione che può essere avviata all'interno del browser dell'utente, eseguendo codice scaricato da un server Web remoto. Tale codice viene eseguito in un'area altamente ristretta (sandbox), che protegge l'utente dalla possibilità che il codice sia malevolo o abbia un comportamento non desiderato; chi pubblica il codice può applicare un certificato che utilizza per firmare digitalmente le applet dichiarandole "sicure", dando loro il permesso di uscire dall'area ristretta e di accedere al file system e alla rete, presumibilmente con l'approvazione e sotto il controllo dell'utente. In realtà le applet non hanno avuto molta fortuna. Infatti il loro impiego presuppone che il client in cui esse vengono eseguite abbia installato il Java

Runtime Environment (JRE) che deve eseguire il codice dell'applet. Hanno avuto fortuna, invece, le applicazioni che prevedono il cosiddetto thin-client, ovvero un client "leggero" che non ha bisogno di particolari strumenti per eseguire il codice remoto (a volte è necessario solo il browser).

Rispetto alla tradizione dei linguaggi a oggetti da cui deriva, Java ha introdotto una serie di importanti novità in merito alla semantica. Fra le più significative si possono citare la possibilità di costruire interfacce grafiche (GUI) con strumenti standard e non proprietari, la possibilità di creare applicazioni multi-thread, ovvero che svolgono in modo concorrente molteplici attività, e il supporto per la riflessione, ovvero la capacità di un programma di agire sulla propria struttura e di utilizzare classi caricate dinamicamente dall'esterno.

Queste peculiarità rendono Java l'alternativa più diffusa sia per la progettazione di applicazioni client/server sia per lo sviluppo di Web Service, anche se l'ambito in cui il linguaggio ha attualmente riscosso maggiore successo è proprio quello Web.

Fra gli argomenti che depongono spesso a favore di Java nella scelta del linguaggio di implementazione di un progetto software moderno, inoltre, si deve certamente considerare la vastità delle librerie standard di cui il linguaggio è dotato e che contribuiscono a renderlo altamente integrabile con le altre tecnologie. Infatti, Java prevede librerie standardizzate per permettere l'accesso alle caratteristiche della macchina (come, ad esempio, grafica e networking) in modo unificato.

Sun Microsystem, che ancora oggi sovrintende alle linee di sviluppo di Java, ha recentemente rilasciato le diverse tecnologie che lo compongono sotto licenza GPL rendendolo un linguaggio di programmazione la cui implementazione di riferimento è libera.

7.2 Best practices considerate per l'attività di code inspection

Nel seguito viene illustrata una serie di "best practices" per lo sviluppo di codice sicuro Java, riconosciute ufficialmente da Sun. Chiaramente tali direttive hanno rappresentato le principali linee guida per l'attività di code inspection con il linguaggio Java da noi effettuata nell'ambito della presente tesi.

7.2.1 Inizializzazioni

Dipendenza dall'inizializzazione

È necessario scrivere le classi in modo tale che l'oggetto sia correttamente inizializzato prima che venga utilizzato. Per evitare l'allocazione di oggetti non inizializzati occorre:

- Rendere tutte le variabili private e, se necessario, fornirne l'accesso dall'esterno della classe stessa; questo deve essere sempre consentito esclusivamente attraverso i metodi `get()` e `set()`.
- Aggiungere in ogni oggetto una variabile booleana privata (ad esempio, `isInitialized`) e fare in modo che ogni costruttore, come ultima operazione, la inizializzi al valore `true`. Il Listato 7.1 mostra l'inizializzazione di questa variabile.

```
public class MyClass {
 private boolean isInitialized;
 private String nome;
 public MyClass(String nome){
  this.nome = nome;
  this.isInitialized = true;
 }
 public String getNome(){
  return (isInitialized == true ? this.nome : null);
 }
}
```

Listato 7.1. Inizializzazione della variabile `isInitialized`

- In ogni metodo che non sia un costruttore verificare che la variabile di inizializzazione della classe sia impostata a `true` prima di eseguire qualsiasi operazione.

Se la classe ha costruttori statici è necessario seguire la stessa procedura, ma a livello di classe; in altre parole occorre:

- Rendere tutte le variabili statiche private e, se necessario, fornirne l'accesso dall'esterno della classe stessa; questo deve sempre essere consentito esclusivamente attraverso i metodi `get()` e `set()`.
- Aggiungere alla classe una variabile booleana privata statica (ad esempio, `isClassInitialized`) e fare in modo che ogni costruttore statico, come ultima operazione, la inizializzi a `true`.
- Prima di eseguire qualsiasi operazione, in ogni metodo statico ed in ogni costruttore, è necessario verificare che la variabile `isClassInitialized` sia impostata a `true`.

Gestione dinamica delle allocazioni/deallocazioni di memoria

Prima di uscire da una classe è sempre necessario azzerare il contenuto delle variabili. La Tabella 7.1 mette in contrapposizione una forma scorretta con una corretta nel caso in cui, ad esempio, la variabile `k` contenga la chiave per decriptare un messaggio cifrato.

Forma Scorretta	*Forma Corretta*
`public class Decodificatore { private byte[] k; }`	`public class Erase { private byte[] k; public void clear() { for(int i = 0; i < k.length; i++) k [i] = (byte) 0x00; } }`

Tabella 7.1. Java: Gestione dinamica delle allocazioni/deallocazioni di memoria

7.2.2 Visibilità

Accesso alle classi, ai metodi ed alle variabili

Le classi, i metodi e le variabili dovrebbero essere definiti come `private` o `protected`. Nei casi in cui ciò non avvenga si deve motivare opportunamente la scelta. Ogni variabile `private` dovrebbe essere accessibile unicamente attraverso metodi `set()` e `get()` per mantenere l'oggetto al sicuro. Il Listato 7.2 illustra un esempio di quanto appena scritto.

```
public class Studente {
 private int eta;
 public int getEta(){
  return this.eta;
 }
 public int setEta(int eta){
  this.eta = eta;
 }
}
```

Listato 7.2. Esempio di visibilità

Ogni costante deve essere definita con i modificatori `static final` per mantenere il suo valore immutabile e per renderla accessibile staticamente. Il Listato 7.3 illustra un esempio di definizione con i suddetti modificatori.

```
static final int key = 1;
```

Listato 7.3. Esempio di modificatore `static final`

Visibilità del package

Le classi, i metodi e le variabili dovrebbero essere esplicitamente marcati come `private`, `protected` o `public` per limitare il livello di accesso da parte di altri

oggetti.

7.2.3 Modificatori

Rendere sempre le classi, i metodi e le variabili di tipo `final`

Le classi, i metodi e le variabili dovrebbero essere definiti come `final`. Nei casi in cui ciò non avvenga si deve motivare opportunamente la scelta. L'utilizzo di tale modificatore consente anche il miglioramento dell'efficienza del programma in fase di esecuzione in quanto non consente il "late binding". I Listati 7.4, 7.5 e 7.6 mostrano alcuni esempi di utilizzo del modificatore `final`.

```
public final class MyFinalClass {
 [...]
}
```

Listato 7.4. Esempio di classe definita `final`

```
public class MyClass {
 final int myConst = 123;
 [...]
}
```

Listato 7.5. Esempio di variabile definita `final`

```
public class MyClass {
 [...]
 public final void stopOverriding() {
  [...]
 }
}
```

Listato 7.6. Esempio di metodo definito `final`

Utilizzo di variabili di tipo `static`

L'utilizzo di variabili di tipo `static` dovrebbe essere evitato per quanto possibile.

7.2.4 Utilizzo degli oggetti mutevoli

Corretto utilizzo degli oggetti mutevoli

Gli oggetti mutevoli (ad esempio, array, liste, vettori, etc.) non dovrebbero mai essere restituiti a codice potenzialmente insicuro e non dovrebbero mai essere memorizzati internamente in modo diretto se provenienti da codice potenzialmente insicuro; essi dovrebbero, invece, essere opportunamente clonati. Le Tabelle 7.2 e 7.3 riportano esempi di utilizzo scorretto e corretto di oggetti mutevoli.

Forma Scorretta	*Forma Corretta*

`public Date getDate() {`
` return fDate;`
`}`
	`public Date getDate() {`
	` return new Date(fDate.getTime());`
	`}`

Tabella 7.2. Java: Un primo esempio dell'utilizzo di oggetti mutevoli

Forma Scorretta	*Forma Corretta*

`public void useDate(Date date) {`
` if (isValid(date))`
` scheduleTask(date);`
`}`
	`public void useDate(Date date) {`
	` Date copied_date = new`
	` Date(date.getTime());`
	` if (isValid(copied_date))`
	` scheduleTask(date);`
	`}`

Tabella 7.3. Java: Un secondo esempio dell'utilizzo di oggetti mutevoli

7.2.5 Definizione delle classi

Evitare l'utilizzo di classi interne

L'utilizzo di "Inner Class" dovrebbe essere sempre evitato. Nei casi del tutto eccezionali, comunque, le classi interne devono sempre essere definite come `private`. La Tabella 7.4 illustra una definizione di classe scorretta ed una corretta.

Forma Scorretta	*Forma Corretta*

`package esempio;`
`public class MyFirstClass {`
` [...]`
` private class MySecondClass {`
` }`
` [...]`
`}`
	`package esempio;`
	`public class MyFirstClass {`
	` [...]`
	`}`
	`class MySecondClass {`
	` [...]`
	`}`

Tabella 7.4. Java: Utilizzo di classi interne

Codice di versione

Ai fini della coerenza di versione sulle classi del package è necessario inserire un codice di versione per ogni classe collocandolo all'interno di una variabile pubblica `final`.

7.2.6 Firma del codice e permessi speciali

Evitare di assegnare al codice permessi speciali

È necessario evitare di firmare il codice prodotto cercando sempre di scriverlo in modo che non abbia bisogno di permessi speciali diversi da quelli definiti nella sandbox. Nei casi del tutto eccezionali in cui il codice necessiti di privilegi speciali al fine di effettuare particolari operazioni (ad esempio, al fine di rilevare le proprietà del sistema, di leggere file anche se collocati in `java.home`, di aprire socket, di scrivere file o di assegnare loro le opportune proprietà, di caricare librerie dinamiche mediante `System.loadLibrary` o `Runtime.getRuntime.loadLibrary`, etc.), è necessario accertarsi che il livello di privilegi concessi sia il minimo indispensabile; è necessario, inoltre, motivare e documentare ampiamente le necessità ed effettuare una verifica approfondita del codice stesso. Nel caso in cui sia necessario firmare il codice prodotto, le classi interessate dovrebbero essere raggruppate in un unico archivio.

7.2.7 Utilizzo dei comandi di sistema

Esecuzione dei comandi di sistema

Supponiamo che un aggressore assegni alla variabile `filename` un valore del tipo:

```
filename ="pietro.txt; /bin/rm -rf /*";
```

La Tabella 7.5 mostra come verrà eseguito il codice malevolo nella forma scorretta; nella forma corretta, invece, tale codice verrà ignorato.

7.2.8 Oggetti

Classi e oggetti non clonabili

Classi ed oggetti non dovrebbero mai essere clonabili. Nel Listato 7.7 viene riportato un esempio di come sia possibile rispettare questa regola.

Forma Scorretta	*Forma Corretta*

`void method (String filename) {`
` System.exec("more " + filename);`
`}`
	`void method (String filename){`
	` if (new File(filename).exists()){`
	`  System.exec("more " + filename);`
	` }`
	`}`

Tabella 7.5. Java: Comandi di sistema

```
public final void clone() throws java.lang.CloneNotSupportedException {
 throw new java.lang.CloneNotSupportedException();
}
[...]
```

Listato 7.7. Java: Esempio di implementazione del metodo `clone()`

Nei casi eccezionali in cui ciò non sia possibile, che dovrebbero essere motivati e ampiamente documentati, è necessario rendere i metodi che consentono la clonazione di tipo `final` in modo da evitare potenziali malevoli override dei metodi stessi. Nel Listato 7.8 viene riportato un esempio di come sia possibile gestire queste eccezioni.

```
public final void clone() throws java.lang.CloneNotSupportedException {
 super.clone();
}
[...]
```

Listato 7.8. Java: Esempio di eccezione relativo al metodo `clone()`

Comparazione degli oggetti

È necessario evitare la comparazione per nome degli oggetti. Nella Tabella 7.6 viene riportato un esempio di come sia possibile rispettare questa regola.

7.2.9 Serializzazione e deserializzazione

Classi e oggetti non serializzabili

Classi ed oggetti non dovrebbero mai essere serializzabili. Nel Listato 7.9 viene riportato un esempio di come sia possibile rispettare questa regola.

Forma Scorretta	*Forma Corretta*

`public class MyClass {`
`public boolean sameClass`
` (Object o) {`
`  Class thisClass = this.getClass();`
`  Class otherClass = o.getClass();`
`  return (thisClass.getName()`
`  == otherClass.getName());`
` }`
`}`
	`package esempio;`
	`public class MyClass {`
	` public boolean sameClass`
	` (Object o) {`
	`  Class thisClass = this.getClass();`
	`  Class otherClass = o.getClass();`
	`  return (thisClass == otherClass);`
	` }`
	`}`

Tabella 7.6. Java: Comparazione degli oggetti

```
private final void writeObject(ObjectOutputStream out) throws
java.io.IOException {
 throw new java.io.IOException("L'oggetto non può essere serializzato ");
}
[...]
```

Listato 7.9. Java: Esempio di oggetti non serializzabili

Classi e oggetti non deserializzabili

Classi ed oggetti non dovrebbero essere mai deserializzabili. Nel Listato 7.10 viene riportato un esempio di come sia possibile rispettare questa regola.

```
private final void readObject(ObjectInputStream in) throws
java.io.IOException {
 throw new java.io.IOException("L'oggetto non può essere deserializzato");
}
[...]
```

Listato 7.10. Java: Esempio di oggetti non deserializzabili

7.2.10 Memorizzazione delle informazioni riservate

Informazioni riservate all'interno del codice

Informazioni riservate, come chiavi crittografiche, password e certificati, non devono mai essere inserite e presenti all'interno del codice o nelle librerie utilizzate.

7.2.11 Package

Creazione dei package

È opportuno creare i package in base alle funzioni e non ai layer dell'applicazione. Già in fase di progettazione dell'applicazione è indispensabile far confluire, nello stesso package, funzioni identiche o simili per genere.

Protezione dei package

È necessario proteggere i package a livello globale, contro l'immissione di codice malevolo o alterato.

A tale proposito si può osservare che l'inserimento nel file `java.security.properties` della seguente linea:

```
package.definition=Pacchetto1 [,Pacccchetto2,...,Pacchetton]
```

causerà un'eccezione nel loader `defineClass` non appena si proverà a definire una nuova classe all'interno del package, a meno che il codice non sia stato dotato del seguente permesso:

```
RuntimePermission("defineClassInPackage."+package)
```

È anche possibile inserire le classi del package in un file JAR sigillato. In questo modo nessun codice potrà ottenere il permesso ad ampliare il package e non ci sarà, quindi, motivo di modificare il file `java.security.properties`.

La protezione dall'accesso può essere ottenuta inserendo la seguente linea nel file `java.security.properties`:

```
package.access=Pacchetto1 [,Pacchetto2,...,Pacchetton]
```

Ciò causerà un'eccezione nel loader `loadClass` non appena si proverà ad accedere ad una classe all'interno del package, a meno che il codice non sia stato dotato del seguente permesso:

```
RuntimePermission("accessClassInPackage."+package)
```

7.2.12 Gestione delle eccezioni

Input nullo

Nel caso in cui un input nullo crei un'eccezione, il programma restituirà un errore sconosciuto. Nella Tabella 7.7 viene mostrato come utilizzare le capacità di logging di Java per mantenere traccia delle eccezioni.

<table>
<tr><th>Forma Scorretta</th><th>Forma Corretta</th></tr>
<tr><td>

```
import java.io.*;
import java.util.*;
public class BadEmptyCatch {
 List quarks = new ArrayList();
 quarks.add("hello word");
 FileOutputStream file = null;
 ObjectOutputStream output = null;
 try{
  file = new
  FileOutputStream("quarks.ser");
  output =
  new ObjectOutputStream(file);
  output.writeObject(quarks);
 }
 catch(Exception exception){
  System.err.println(exception);
 }
 finally{
  try {
   if (output != null) {
    output.close();
   }
  }
  catch(Exception exception){
  }
 }
}
```

</td><td>

```
import java.io.*;
import java.util.*;
import java.util.logging.*;
public class ExerciseSerializable {
 public static void main(String args) {
  List quarks = new ArrayList();
  quarks.add("hello word");
  ObjectOutput output = null;
  try{
   OutputStream file = new
   FileOutputStream( "quarks.ser");
   OutputStream buffer = new
   BufferedOutputStream( file );
   output =
   new ObjectOutputStream(buffer);
   output.writeObject(quarks);
  }
  catch(IOException ex){
   fLogger.log(Level.SEVERE,
   "Cannot perform output.", ex);
  }
  finally{
   try {
    if (output != null) {
     output.close();
    }
   }
   catch (IOException ex ){
    fLogger.log(Level.SEVERE,
    "Cannot close output stream.", ex);
   }
  }
 }
}
```

</td></tr>
</table>

Tabella 7.7. Java: Gestione delle eccezioni nel caso di input nullo

Passare tutti i dati pertinenti alle eccezioni

Nel caso si manifesti un'eccezione è obbligatorio far si che tutti i dati pertinenti vengano passati al costruttore dell'eccezione stessa. Nel Listato 7.11 è riportato un esempio relativo a quanto appena scritto.

```
public final class RangeChecker {
 static public boolean isInRange( int aNumber, int aLow, int aHigh ){
  if (aLow > aHigh) {
   throw new IllegalArgumentException("Low:" + aLow + " greater than
   High:" + aHigh);
  }
  return (aLow <= aNumber && aNumber <= aHigh);
 }
}
```

Listato 7.11. Java: Passaggio al costruttore dei dati pertinenti

Specificare la clausola throws

È opportuno evitare di raggruppare le eccezioni in una classe di eccezioni generica in quanto ciò rappresenterebbe una perdita di informazioni importanti. La Tabella 7.8 illustra un esempio di forma scorretta e la corrispondente forma corretta.

Forma Scorretta	*Forma Corretta*

`import java.io.*;`
`import java.util.*;`
`public class BadGenericThrow {`
` public void makeFile() throws`
` Exception {`
`  //create a Serializable List`
`  List<String> quarks = new`
`  ArrayList<String>();`
`  quarks.add("hello word");`
`  FileOutputStream file = null;`
`  ObjectOutputStream output = null;`
`  try{`
`   file = new`
`   FileOutputStream("quarks.ser");`
`   output = new`
`   ObjectOutputStream(file);`
`   output.writeObject(quarks);`
`  }`
`  finally{`
`   if (output != null) {`
`    output.close();`
`   }`
`  }`
` }`
`}`
	`import java.io.*;`
	`import java.util.*;`
	`public class BadGenericThrow {`
	` public void makeFile() throws`
	` IOException, FileNotFoundException{`
	`  //create a Serializable List`
	`  List<String> quarks = new`
	`  ArrayList<String>();`
	`  quarks.add("hello word");`
	`  FileOutputStream file = null;`
	`  ObjectOutputStream output = null;`
	`  try{`
	`   file = new`
	`   FileOutputStream("quarks.ser");`
	`   output = new`
	`   ObjectOutputStream(file);`
	`   output.writeObject(quarks);`
	`  }`
	`  finally{`
	`   if (output != null) {`
	`    output.close();`
	`   }`
	`  }`
	` }`
	`}`

Tabella 7.8. Java: Utilizzo della clausola throws

7.3 Applet Java

Per la scrittura delle Applet Java devono essere rispettate tutte le regole descritte nella precedente sezione e, in aggiunta ad esse, quelle di seguito riportate.

7.3.1 Informazioni critiche

Non si devono mai memorizzare nel codice informazioni critiche per l'azienda, cioè informazioni relative ad aspetti di sicurezza (ad esempio, password, chiavi crittografiche, etc.) e che possono essere in qualche modo nocive. Questa regola è valida anche se vengono utilizzati meccanismi di offuscamento o crittografia delle Applet.

7.3.2 Visibilità

Le classi, i metodi o le variabili devono essere dichiarati di tipo `private` a meno che non vi sia una buona ragione per non farlo; in quest'ultimo caso la scelta deve essere ampiamente documentata ed approvata.

7.3.3 Overriding

A meno che non vi sia una valida ragione, non si deve mai effettuare l'overriding dei metodi di gestione dei thread (ad esempio, dei metodi `wait()`, `notify()`, etc. della classe `java.lang.Thread`).

7.3.4 Modificatori

Tutti i metodi delle classi devono essere definiti di tipo `final` a meno che non vi sia una buona ragione per non farlo; in quest'ultimo caso la scelta deve essere ampiamente documentata ed approvata.

7.3.5 Firma delle Applet

Il codice di un'Applet deve essere firmato solo se assolutamente necessario. In tal caso il certificato utilizzato deve essere valido in tutte le sue parti, deve essere emesso da un'autorità di certificazione riconosciuta a livello internazionale e deve essere sempre verificabile dal browser che esegue l'Applet. Certificati temporanei e di test non dovrebbero mai essere utilizzati né in ambiente di collaudo né in ambiente di esercizio.

7.3.6 Validazione delle informazioni trasferite

Tutte le informazioni, trasferite in qualsiasi modo e con qualsiasi protocollo, da un'Applet verso una componente server devono sempre essere validate anche lato server.

7.3.7 Utilizzo delle Applet di terze parti

Non devono mai essere utilizzate Applet di dubbia o non certificata provenienza e/o Applet per le quali non si dispone del relativo codice sorgente. In ogni caso le Applet possono essere utilizzate, negli ambienti di sviluppo, di collaudo e di esercizio, esclusivamente previa analisi del codice relativo e previa ricompilazione andata a buon fine e senza avvertimenti.

7.4 Servlet Java

7.4.1 Utilizzo delle richieste di tipo HTTP POST e HTTP GET

L'invio di informazioni tramite form HTML deve avvenire esclusivamente tramite richieste di tipo HTTP POST e, pertanto, l'implementazione del metodo `doGet()` delle Servlet deve contenere soltanto la corrispondente e corretta gestione dell'errore.

7.4.2 Filtraggio dell'input

Qualsiasi parte di una richiesta HTTP non validata da un sistema di controllo lato server è pericolosa dal punto di vista della sicurezza. È necessario, quindi, assicurarsi che i parametri di una richiesta HTTP vengano validati prima di un loro effettivo utilizzo. Tali informazioni (ad esempio, URL, cookie, form field, hidden field, header, etc., ottenuti dai metodi `getParameter()`, `getCookie()`, o `getHeader()` degli oggetti `HttpServletReques`), prima di essere utilizzate, devono essere rigorosamente validate lato server e ne deve essere effettuata la canonicalizzazione, cioè la semplificazione della codifica.

Ogni informazione in input, intesa in questo caso come singolo parametro, deve essere analizzata in modo tale da specificare quale tipo di dato possa essere ammesso in ingresso. I parametri da controllare in fase di validazione sono:

- il tipo di dato (string, integer, real, etc.);
- il set di caratteri consentito;
- la lunghezza minima e massima;
- la possibilità o meno di utilizzare il tipo NULL;
- la richiesta o meno del parametro;
- la possibilità o meno di avere duplicati;
- l'intervallo numerico;
- i valori ammessi;
- i pattern.

Inoltre, tali informazioni devono essere accuratamente scansionate con l'obiettivo di ricercare qualsiasi carattere speciale e sostituirlo con il corrispondente carattere HTML. Tale sostituzione deve essere sempre eseguita all'interno dei metodi `doGet()` e `doPost()` di tutte le Servlet che compongono l'applicazione Web. La Tabella 7.9 mostra un esempio dei caratteri speciali che devono essere ricercati e le corrispondenti sostituzioni HTML che devono essere effettuate.

Carattere Speciale	*Carattere HTML sostitutivo corrispondente*
`<`	`<`
`>`	`>`
`(`	`&40;`
`)`	`&41;`
`#`	`&35;`
`&`	`&38;`

Tabella 7.9. Sostituzioni dei caratteri speciali da effettuare nel codice HTML

I Web application server più evoluti mettono a disposizione funzioni native che svolgono questa operazione (ad esempio, `weblogic.servlet.security.Utils-.encodeXSS()` di BEA WebLogic Server). Se si utilizza uno di questi Web application server è sempre preferibile utilizzare queste funzioni interne piuttosto che creare funzioni esterne.

Se si utilizza BEA WebLogic Server e la variabile `user_input` contiene le informazioni inserite dall'utente tramite un form HTML, è necessario sostituire la seguente istruzione:

```
out.print("User input: " + user_input);
```

con l'istruzione:

```
out.print(("User input: " + weblogic.security.servlet.encodeXSS(user_input) + "!");
```

È fortemente raccomandato accettare in input informazioni composte esclusivamente da caratteri ammissibili per il contesto dell'informazione stessa. Ad esempio, se un utente sta trasferendo l'informazione sulla sua età tramite un form HTML, i caratteri che possono essere accettati sono soltanto cifre (0-9); non c'è alcuna ragione che trasferisca qualsiasi altro carattere diverso e tanto meno caratteri speciali.

7.4.3 Utilizzo dei Web Application Firewall

In tutti i casi in cui sia possibile è consigliabile l'integrazione di *Web Application Firewall* che forniscono meccanismi di validazione dell'input e di protezione per tutti i tipi di dati ricevuti da una richiesta HTTP, inclusi gli URL, i form, i cookie, le query string, gli hidden field e i parametri.

7.4.4 Proteggere i dati personali e/o sensibili

Tutte le transazioni che prevedono la ricezione, da parte di una Servlet, di dati personali e/o sensibili (ad esempio, login, password, fede religiosa, malattie, etc.) devono sempre avvenire in modo sicuro e, almeno, su protocollo HTTP protetto tramite SSL (HTTPS). In quest'ottica il processo di autenticazione e di autorizzazione di un utente deve sempre avvenire tramite protocollo HTTPS.

7.4.5 Switching tra le modalità SSL e non-SSL

Per le risorse Web che devono essere protette tramite SSL non deve mai essere consentito lo switch in modalità non-SSL. Tali risorse devono essere accessibili esclusivamente tramite protocollo HTTPS e mai tramite il protocollo HTTP non protetto. Ciò può essere ottenuto utilizzando un "Servlet Filter" che rifiuta ogni richiesta di accesso, in modalità non-SSL, alle risorse protette.

7.4.6 Statement SQL

Se una Servlet accede ad un database, i relativi comandi SQL non devono mai essere costruiti utilizzando concatenazioni di stringhe, né tanto meno concatenazioni di informazioni inserite dagli utenti, anche se verificate e validate.

A tale proposito, è necessario utilizzare l'interfaccia PreparedStatement che, tramite il driver JDBC, effettua la canonicalizzazione dei parametri in modo automatico. Il Listato 7.12 riporta un esempio di utilizzo dell'interfaccia PreparedStatement.

```
. . .
String selectStatement = "SELECT * FROM User WHERE userId = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
. . .
```

Listato 7.12. Java Servlet: Utilizzo dell'interfaccia PreparedStatement

7.4.7 Token di sessione

Si deve evitare di creare token di sessione specifici; è opportuno, infatti, utilizzare preferibilmente quelli messi a disposizione del Web container (o Web application server) in uso, gestendo le sessioni utente tramite l'apposita interfaccia javax.servlet.http.HttpSession. In ogni caso, i token di sessione dovrebbero sempre rispettare le seguenti regole:

- non devono mai essere inclusi nelle URL;
- devono essere catene lunghe e complicate di numeri casuali che non possano essere facilmente indovinati;
- devono cambiare frequentemente durante una sessione;
- devono cambiare quando si passa ad utilizzare protocolli come SSL;
- non devono mai essere scelti dall'utente.

7.4.8 Cookie

L'utilizzo dei cookie con le Servlet deve sempre rispettare le seguenti regole minime:

- I cookie non devono essere mai utilizzati per memorizzare dati personali o informazioni sensibili.
- Prima di trasferire informazioni personali o sensibili verso un utente è necessario richiedere sempre la sua autenticazione ed autorizzazione tramite inserimento di login e password. Non ci si deve mai basare sulla presenza o meno di un cookie precedentemente memorizzato.
- Si devono configurare i cookie di sessione in modo tale che scadano quando l'utente chiude il browser.
- Ci si deve assicurare che tutte le informazioni contenute nei cookie siano accuratamente verificate e filtrate prima di essere utilizzate e/o inserite nei documenti HTML.

7.4.9 Limitare la dimensione delle risposte HTTP

In tutti i casi in cui sia possibile è opportuno limitare la lunghezza delle risposte HTTP alla massima dimensione consentita, troncando quelle che hanno una dimensione eccedente.

7.4.10 HTTP Referer

In tutti i casi in cui sia possibile, è opportuno verificare il campo "Referer" dell'intestazione HTTP e rigettare le informazioni provenienti da host o link incorretti e/o inaspettati.

7.4.11 Trattamento dei file e degli oggetti embedded

Una Servlet non deve mai accettare in input contenuti inviati da un utente che contengano tag HTML tipici dell'inclusione di file, oppure oggetti come `<EMBED>`, `<OBJECT>` e `<SCRIPT>`.

7.4.12 Corretta gestione degli errori e delle eccezioni

Tutte le eccezioni che si verificano durante l'esecuzione delle Servlet che costituiscono l'applicazione Web devono essere catturate e gestite opportunamente. I relativi messaggi di errore sollevati (ad esempio, "database dump", codici di errore, "NULL pointer exception", "system call failure", "unavailable database", "network timeout", etc.), devono essere visualizzati in accordo con uno schema ben dettagliato; più specificatamente, agli utenti generici devono essere inviate le informazioni minime in grado di aiutarli nella comprensione degli errori stessi (senza rivelare dettagli superflui) mentre le informazioni sulla diagnostica devono essere inviate per la visualizzazione esclusivamente agli amministratori dell'applicazione stessa.

Il meccanismo di gestione degli errori deve essere in grado di gestire ogni tipo di dati in ingresso e, nel frattempo, garantire la sicurezza. Devono essere previsti dei messaggi di errore semplici, in grado di indicare la causa e di archiviare i tentativi di intrusione in modo tale da poterli verificare in un secondo tempo.

La gestione degli errori non deve essere concentrata soltanto sui dati forniti in ingresso dall'utente, ma deve includere anche tutti gli errori che possono essere generati da componenti interni come system call, query sul database o altre funzioni interne.

7.4.13 Limitare l'utilizzo delle risorse macchina

In tutti i casi in cui sia possibile è opportuno implementare meccanismi che consentano di limitare al massimo il numero di risorse allocate per ogni singolo utente.

Per gli utenti autenticati è possibile stabilire una quota in modo da poter limitare il carico massimo che un utente può applicare al sistema.

Per gli utenti non autenticati, si dovrebbero evitare tutti gli accessi al database o ad altre applicazioni avide di risorse ritenute superflue, mantenendo, ad esempio, in una cache il contenuto dei dati ricevuti da questi utenti, invece di eseguire delle query direttamente sul database.

7.5 Vulnerabilità per il linguaggio di programmazione Java riscontrabili tramite Fortify

Come detto in precedenza, per la nostra attività di code inspection abbiamo utilizzato il software Fortify Source Code Analyzer.

Tale software, per quanto riguarda il linguaggio di programmazione Java, è in grado di riscontrare una serie di vulnerabilità, ciascuna legata a un determinato dominio.

Più specificatamente, le vulnerabilità che è possibile riscontrare con il software Fortify Source Code Analyzer sono le seguenti:

- *Access Control: Database*. Questa vulnerabilità riguarda il fatto che, senza un opportuno controllo degli accessi, l'esecuzione di un comando SQL contenente una chiave primaria controllata dall'utente può consentire ad un attaccante di visualizzare record senza autorizzazione. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Code Correctness: Call to System.gc()*. Questa vulnerabilità riguarda il fatto che le esplicite richieste di "garbage collection" sono spesso sintomo di problemi prestazionali. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Code Correctness: Call to Thread.run()*. Questa vulnerabilità riguarda il fatto che il programma chiama un metodo `run()` del thread invece di chiamare `start()`. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Class Does Not Implement Cloneable*. Questa vulnerabilità riguarda il fatto che una classe implementa un metodo `clone()`, ma non implementa `Cloneable`. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Double-Checked Locking*. Questa vulnerabilità riguarda il fatto che "Double-checked locking" è un modo di dire che non realizza l'effetto desiderato. Il dominio di riferimento per questa vulnerabilità è "Time and State".

- *Code Correctness: Erroneous Class Compare*. Questa vulnerabilità riguarda il fatto che la determinazione di un tipo di oggetto basato sul nome della sua classe può provocare un comportamento inaspettato o consentire ad un attaccante l'introduzione di una classe malevola. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Erroneous () Method*. Questa vulnerabilità si verifica quando il metodo `()` non chiama `super.()`. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Code Correctness: Erroneous String Compare*. Questa vulnerabilità riguarda il fatto che le stringhe dovrebbero essere comparate utilizzando il metodo `equals()` anziché "`==`" oppure "`!=`". Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: Misspelled Method Name*. Questa vulnerabilità riguarda il tentativo di sovrascrittura di un metodo comune Java, ma probabilmente non ha l'effetto desiderato. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Code Correctness: NULL Argument to equals()*. Questa vulnerabilità riguarda il fatto che l'espressione `obj.equals(NULL)` dovrebbe essere sempre falsa. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Command Injection (Data Flow)*. Questa vulnerabilità riguarda il fatto che comandi generati da fonti o ambienti non fidati possono consentire l'esecuzione di codice e comandi malevoli. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Command Injection (Semantic)*. Questa vulnerabilità riguarda il fatto che l'esecuzione di comandi che includono input utente non validato può far si che un'applicazione esegua comandi a beneficio di un attaccante. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Cross-Site Scripting*. Questa vulnerabilità riguarda il fatto che l'invio di dati non validati ad un browser Web potrebbe portare all'esecuzione di codice malevolo. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Cross-Site Scripting: Poor Validation*. Questa vulnerabilità riguarda il fatto che fare affidamento sulla codifica XML per validare l'input utente può provocare l'esecuzione di codice malevolo nel browser. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Dead Code: Expression is Always False*. Questa vulnerabilità si verifica quando un'espressione sarà sempre considerata falsa. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Dead Code: Expression is Always True*. Questa vulnerabilità si verifica quando un'espressione sarà sempre considerata vera. Il dominio di riferimento per questa vulnerabilità è "Code Quality".

- *Dead Code: Unused Field*. Questa vulnerabilità si verifica quando un campo non è mai utilizzato. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Dead Code: Unused Method*. Questa vulnerabilità si verifica quando un metodo non è raggiungibile da nessun metodo esterno alla classe. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Denial of Service*. Questa vulnerabilità riguarda il fatto che un attaccante potrebbe causare il blocco di un programma oppure renderlo non disponibile agli utenti autorizzati. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *EJB Bad Practices: Use of AWT/Swing*. Questa vulnerabilità riguarda il fatto che il programma viola la specifica "Enterprise JavaBeans" utilizzando AWT/Swing. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *EJB Bad Practices: Use of Class Loader*. Questa vulnerabilità riguarda il fatto che il programma viola la specifica "Enterprise JavaBeans" utilizzando la classe loader. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *EJB Bad Practices: Use of java.io*. Questa vulnerabilità riguarda il fatto che il programma viola la specifica "Enterprise JavaBeans" utilizzando il package `java.io` . Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *EJB Bad Practices: Use of Sockets*. Questa vulnerabilità riguarda il fatto che il programma viola la specifica "Enterprise JavaBeans" utilizzando i socket. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *EJB Bad Practices: Use of Synchronization Primitives*. Questa vulnerabilità riguarda il fatto che il programma viola la specifica "Enterprise JavaBeans" utilizzando le primitive di sincronizzazione dei thread. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *HTTP Response Splitting*. Questa vulnerabilità riguarda il fatto che l'inserimento di dati HTTP non validati in un header di risposta può causare cache-poisoning, cross-site scripting e attacchi di tipo page hijacking. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Insecure Randomness*. Questa vulnerabilità riguarda il fatto che i generatori standard di numeri pseudo-casuali non sono in grado di resistere ad attacchi crittografici. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *J2EE Bad Practices: getConnection()*. Questa vulnerabilità riguarda il fatto che lo standard J2EE proibisce la gestione diretta delle connessioni. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *J2EE Bad Practices: Leftover Debug Code*. Questa vulnerabilità riguarda il fatto che il codice di debug può creare punti di ingresso nelle applicazioni Web in esercizio. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *J2EE Bad Practices: Non-Serializable Object Stored in Session*. Questa vulnerabilità riguarda il fatto che la memorizzazione di un oggetto non serializ-

zabile come un attributo `HttpSession` può compromettere l'affidabilità dell'applicazione. Il dominio di riferimento per questa vulnerabilità è "Time and State".

- *J2EE Bad Practices: Sockets.* Questa vulnerabilità riguarda il fatto che l'utilizzo di socket all'interno di applicazioni Web può facilmente introdurre problematiche di sicurezza. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *J2EE Bad Practices: System.exit().* Questa vulnerabilità riguarda il fatto che un'applicazione Web non dovrebbe in nessun caso tentare di terminare il proprio container. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *J2EE Bad Practices: Threads.* Questa vulnerabilità riguarda il fatto che in alcune circostanze la gestione dei thread è vietata nelle applicazioni Web e può sempre favorire la presenza di errori. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *J2EE Misconfiguration: Debug Information.* Questa vulnerabilità riguarda il fatto che, in Tomcat, un livello di debug uguale o maggiore di 3 può causare l'inserimento nei log di informazioni sensibili, comprese le password. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Incomplete Error Handling (Missing 404).* Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere la corretta gestione delle pagine di errore 404 al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Incomplete Error Handling (Missing 500).* Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere la corretta gestione delle pagine di errore 500 al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Incomplete Error Handling (Missing Throwable).* Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere la corretta gestione delle pagine di errore `java.lang.Throwable` al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Insecure Transport.* Questa vulnerabilità riguarda il fatto che l'applicazione dovrebbe utilizzare SSL per l'accesso a tutte le pagine protette. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Insufficient Session-ID Length.* Questa vulnerabilità riguarda il fatto che gli identificatori di sessione devono avere una lunghezza minima di 24 byte al fine di prevenire attacchi di tipo "brute force". Il dominio di riferimento per questa vulnerabilità è "Environment".

- *J2EE Misconfiguration: Missing Error Handling (Missing 404)*. Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere delle pagine di errore 404 al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Missing Error Handling (Missing 500)*. Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere delle pagine di errore 500 al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Missing Error Handling (Missing Throwable)*. Questa vulnerabilità riguarda il fatto che un'applicazione Web deve prevedere delle pagine di errore `java.lang.Throwable` al fine di prevenire la rivelazione di informazioni di sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Unsafe Bean Declaration*. Questa vulnerabilità riguarda il fatto che gli "entity bean" non dovrebbero essere dichiarati in remoto. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *J2EE Misconfiguration: Weak Access Permissions*. Questa vulnerabilità riguarda il fatto che il permesso di invocare metodi EJB non dovrebbe essere consentito a `ANYONE`. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *JavaScript Hijacking: Vulnerable Framework (GWT)*. Questa vulnerabilità riguarda il fatto che le applicazioni che si basano sul framework Google Web Toolkit (GTW) Ajax potrebbero essere vulnerabili al "JavaScript hijacking" che consente ad utenti non autorizzati di leggere dati confidenziali. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *JavaScript Hijacking: Vulnerable Framework (DWR)*. Questa vulnerabilità riguarda il fatto che le applicazioni che si basano sul framework Direct Web Remoting (DWR) Ajax 1.1.4 e precedenti versioni sono vulnerabili al "JavaScript hijacking" che consente ad utenti non autorizzati di leggere dati confidenziali. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *JavaScript Hijacking: Ad Hoc Ajax*. Questa vulnerabilità riguarda il fatto che le applicazioni che utilizzano la notazione JavaScript per il trasporto di informazioni sensibili possono essere vulnerabili al "JavaScript hijacking", che può causare l'accesso ad informazioni riservate. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Log Forging*. Questa vulnerabilità riguarda il fatto che la scrittura dei file di tracciamento da parte di codice utente non validato potrebbe consentire ad un attaccante di falsificare i dati di accesso al sistema o di inserire contenuto malevolo nei dati di tracciamento. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Missing Check against NULL*. Questa vulnerabilità riguarda il fatto che è sempre necessario verificare la corretta gestione delle funzioni che potrebbe-

ro restituire NULL come valore di ritorno ad una chiamata. Il dominio di riferimento per questa vulnerabilità è "API Abuse".

- *Missing Check for NULL Parameter*. Questa vulnerabilità si verifica quando una funzione non confronta il suo parametro con NULL. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Missing XML Validation*. Questa vulnerabilità riguarda il fatto che la non corretta valutazione di dati XML può consentire ad un attaccante di inviare input malevolo. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *NULL Dereference*. Questa vulnerabilità riguarda il fatto che la gestione scorretta dei puntatori a NULL potrebbe causare un `NULLPointerException`. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Object Model Violation: Erroneous clone() Method*. Questa vulnerabilità riguarda il fatto che il metodo dovrebbe chiamare `super.clone()` per ottenere un nuovo oggetto. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Object Model Violation: Just One of equals() and hashCode() Defined*. Questa vulnerabilità si verifica quando una classe sovrascrive solo una tra i metodi `equals()` e `hashCode()`. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Obsolete*. Questa vulnerabilità riguarda il fatto che l'utilizzo di funzioni deprecate o antiquate potrebbe evidenziare un cattivo codice. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Often Misused: Authentication*. Questa vulnerabilità riguarda il fatto che un attaccante potrebbe aver ottenuto il controllo di un server DNS. È necessario non validare intrinsecamente le informazioni ottenute. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Password Management*. Questa vulnerabilità riguarda il fatto che il salvataggio delle password in testo chiaro potrebbe compromettere l'intero sistema. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Password Management: Empty Password in Configuration File*. Questa vulnerabilità riguarda il fatto che l'utilizzo di una stringa vuota come password non è sicuro. Il dominio di riferimento per questa vulnerabilità è "Environment".
- *Password Management: Hardcoded Password*. Questa vulnerabilità riguarda il fatto che le password hardcoded possono compromettere seriamente la sicurezza del sistema. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Password Management: Password in Configuration File*. Questa vulnerabilità riguarda il fatto che il salvataggio delle password in testo chiaro in un file di configurazione potrebbe compromettere l'intero sistema. Il dominio di riferimento per questa vulnerabilità è "Environment".

- *Password Management: Password in Redirect.* Questa vulnerabilità riguarda il fatto che l'invio di una password come parte di reindirizzamento HTTP potrebbe causarne la visualizzazione, la registrazione su file o la memorizzazione in una cache. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Password Management: Weak Cryptography.* Questa vulnerabilità riguarda il fatto che offuscare una password con una codifica debole non protegge la password stessa. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Path Manipulation.* Questa vulnerabilità riguarda il fatto che consentire all'input utente il controllo dei percorsi usati nelle operazioni sul file system potrebbe permettere ad un attaccante di accedere o modificare le risorse protette di sistema. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Poor Error Handling: Empty Catch Block.* Questa vulnerabilità riguarda il fatto che ignorare un'eccezione potrebbe far sfuggire al programma stati e condizioni inaspettati. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Error Handling: Overly Broad Catch.* Questa vulnerabilità si verifica quando un catch block gestisce un'ampia gamma di eccezioni, intrappolando potenzialmente diversi problemi che non dovrebbero essere trattati in quel punto del programma. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Error Handling: Overly Broad Throws.* Questa vulnerabilità riguarda il fatto che il metodo invoca una generica eccezione che rende difficile ai chiamanti la gestione e il recupero degli errori. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Error Handling: Program Catches NULLPointerException.* Questa vulnerabilità riguarda il fatto che, in genere, è una cattiva pratica catturare le `NULLPointerException`. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Error Handling: Return Inside Finally.* Questa vulnerabilità riguarda il fatto che il ritorno dall'interno di un finally block causerà la perdita di eccezioni. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Error Handling: Unhandled SSL Exception.* Questa vulnerabilità riguarda il fatto che il fallimento nel gestire esplicitamente le eccezioni SSL può causare la perdita, da parte dell'applicazione, di stati e condizioni inaspettati. Il dominio di riferimento per questa vulnerabilità è "Errors".
- *Poor Logging Practice: Logger Not Declared Static Final.* Questa vulnerabilità riguarda il fatto che i logger dovrebbero essere dichiarati come `static` e `final`. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".

- *Poor Logging Practice: Multiple Loggers*. Questa vulnerabilità riguarda il fatto che è opportuno ridurre al minimo l'uso di logger differenti, prediligendo l'impiego di diversi livelli di verbosità. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Poor Logging Practice: Use of a System Output Stream*. Questa vulnerabilità riguarda il fatto che l'utilizzo di `System.out` o `System.err` rende più complesso monitorare il comportamento di un applicativo. Il dominio di riferimento per questa vulnerabilità è " Encapsulation".
- *Poor Style: Confusing Naming (Class and Member)*. Questa vulnerabilità riguarda il fatto che un membro della classe ha lo stesso nome della classe di chiusura. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Confusing Naming (Field and Method)*. Questa vulnerabilità riguarda il fatto che la classe contiene un campo e un metodo con lo stesso nome. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Empty Synchronized Block*. Questa vulnerabilità si verifica quando un blocco sincronizzato è vuoto; è improbabile che la sincronizzazione realizzi l'effetto desiderato. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Explicit Call to ()*. Questa vulnerabilità riguarda il fatto che il metodo `()` dovrebbe essere chiamato esclusivamente dalla JVM dopo aver provveduto al garbage collection. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Poor Style: Identifier Contains Dollar Symbol ($)*. Questa vulnerabilità riguarda il fatto che non è opportuno l'utilizzo del simbolo del dollaro ($) come parte di un identificatore. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Redundant Initialization*. Questa vulnerabilità riguarda il fatto che l'assegnazione di un valore ad una variabile che non viene mai utilizzata è uno spreco. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Poor Style: Value Never Read*. Questa vulnerabilità riguarda il fatto che è uno spreco assegnare un valore ad una variabile che non viene mai letta. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Privacy Violation*. Questa vulnerabilità riguarda il fatto che il trattamento scorretto di informazioni private, come le password dei clienti o i codici fiscali, potrebbe compromettere la privacy degli utenti ed è spesso illegale. Il dominio di riferimento per questa vulnerabilità è "Security Features".
- *Process Control (Data Flow)*. Questa vulnerabilità riguarda il fatto che consentire il caricamento di librerie esterne da una fonte non validata potrebbe permettere l'esecuzione di codice arbitrario. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Process Control (Semantic)*. Questa vulnerabilità riguarda il fatto che consentire il caricamento di librerie senza la specificazione di un percorso assoluto

potrebbe permetterne la loro sostituzione con librerie arbitrarie da parte di un attaccante. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".

- *Race Condition: Singleton Member Field.* Questa vulnerabilità riguarda il fatto che i campi di una Servlet potrebbero consentire a un utente di accedere ai dati appartenenti ad un altro. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *Race Condition: Static Database Connection.* Questa vulnerabilità riguarda il fatto che le connessioni a database, memorizzate in campi statici, vengono condivise tra diversi thread. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *Resource Injection.* Questa vulnerabilità riguarda il fatto che permettere ad input utente il controllo di identificatori di risorse può consentire, in alcune circostanze, ad un attaccante di accedere o modificare risorse protette di sistema. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Session Fixation.* Questa vulnerabilità riguarda il fatto che l'autenticazione di un utente senza aver prima invalidato ogni identificatore di sessione fornisce ad un attaccante l'opportunità di appropriarsi di sessioni autenticate. Il dominio di riferimento per questa vulnerabilità è "Time and State".
- *Setting Manipulation.* Questa vulnerabilità riguarda il fatto che consentire la modifica dei parametri di funzionamento potrebbe causare un blocco del servizio o comportamenti imprevisti dell'applicazione. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *SQL Injection.* Questa vulnerabilità riguarda il fatto che consentire ad un utente la costruzione di un'espressione dinamica SQL potrebbe permettere ad un attaccante di modificare opportunamente l'espressione per eseguire comandi arbitrari. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *SQL Injection: Hibernate.* Questa vulnerabilità riguarda il fatto che l'utilizzo di `Hibernate` per eseguire un'espressione dinamica SQL costruita a partire da input utente può permettere ad un attaccante la modifica del significato dell'espressione o l'esecuzione di comandi SQL arbitrari. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Duplicate Validation Forms.* Questa vulnerabilità riguarda il fatto che diversi form di validazione con lo stesso nome indicano che la logica di validazione non è aggiornata. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Erroneous validate() Method.* Questa vulnerabilità si verifica quando un form di validazione definisce un metodo `validate()`, ma fallisce la chiamata di `super.validate()`. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".

- *Struts: Form Does Not Extend Validation Class.* Questa vulnerabilità riguarda il fatto che tutti i moduli Struts dovrebbero estendere una classe `Validator`. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Form Field Without Validator.* Questa vulnerabilità riguarda il fatto che tutti i campi di un form devono essere validati. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Plugin Framework Not In Use.* Questa vulnerabilità riguarda il fatto che è consigliato l'utilizzo del validatore Struts al fine di prevenire vulnerabilità derivanti da input non verificato. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Unused Validation Form.* Questa vulnerabilità riguarda il fatto che un form di validazione non utilizzato indica che la logica di validazione non è aggiornata. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Unvalidated Action Form.* Questa vulnerabilità riguarda il fatto che ogni form di azione dovrebbe avere un corrispondente form di validazione. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Validator Turned Off.* Questa vulnerabilità si verifica quando il rilevamento di un form di azione disabilita il metodo `validate()` del form. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Struts: Validator Without Form Field.* Questa vulnerabilità riguarda il fatto che i campi di validazione che non compaiono nei form corrispondenti indicano che la logica di validazione è obsoleta. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *System Information Leak.* Questa vulnerabilità riguarda il fatto che rivelare dati di sistema o informazioni di debug può favorire i piani di attacco di un malintenzionato. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *System Information Leak: HTML Comment in JSP.* Questa vulnerabilità riguarda il fatto che la presenza di commenti HTML all'interno di codice JSP può fornire informazioni sul sistema utilizzato, aumentando le possibilità di successo di attacchi. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *System Information Leak: Missing Catch Block.* Questa vulnerabilità riguarda il fatto che se una Servlet non gestisce correttamente tutte le eccezioni, può rivelare informazioni che facilitino eventuali attacchi. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Trust Boundary Violation.* Questa vulnerabilità riguarda il fatto che l'inserimento nella stessa struttura di dati fidati e di dati non fidati può portare

all'utilizzo erroneo di dati non validati. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".

- *Unchecked Return Value*. Questa vulnerabilità riguarda il fatto che la mancata considerazione di un valore restituito da un metodo potrebbe portare il programma a lasciarsi sfuggire stati e condizioni imprevisti. Il dominio di riferimento per questa vulnerabilità è "API Abuse".
- *Unreleased Resource*. Questa vulnerabilità riguarda il fatto che il programma potrebbe fallire il rilascio di risorse di sistema. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unreleased Resource: Database*. Questa vulnerabilità riguarda il fatto che il programma potrebbe fallire il rilascio di risorse relative alla base di dati. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unreleased Resource: Stream*. Questa vulnerabilità riguarda il fatto che il programma potrebbe fallire il rilascio di risorse relative agli Stream. Il dominio di riferimento per questa vulnerabilità è "Code Quality".
- *Unsafe JNI*. Questa vulnerabilità riguarda il fatto che l'utilizzo improprio della Java Native Interface (JNI) potrebbe esporre le applicazioni Java a falle di sicurezza presenti in altri linguaggi. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".
- *Unsafe Mobile Code: Access Violation*. Questa vulnerabilità si verifica quando il programma viola i principi di sicurezza del codice "mobile" restituendo un array privato da un metodo di pubblico accesso. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Unsafe Mobile Code: Inner Class*. Questa vulnerabilità si verifica quando il programma viola i principi di sicurezza del codice "mobile" utilizzando una inner class. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Unsafe Mobile Code: Public () Method*. Questa vulnerabilità si verifica quando il programma viola i principi di sicurezza del codice "mobile" dichiarando `public` un metodo `()`. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Unsafe Mobile Code: Unsafe Array Declaration*. Questa vulnerabilità si verifica quando il programma viola i principi di sicurezza del codice "mobile" dichiarando un array `public`, `final` e `static`. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Unsafe Mobile Code: Unsafe Public Field*. Questa vulnerabilità si verifica quando il programma viola i principi di sicurezza del codice "mobile" dichiarando una variabile "member" `public`, ma non `final`. Il dominio di riferimento per questa vulnerabilità è "Encapsulation".
- *Unsafe Reflection*. Questa vulnerabilità riguarda il fatto che un attaccante potrebbe essere in grado di modificare il percorso del flusso di controllo del-

l'applicazione aggirando i controlli di sicurezza. Il dominio di riferimento per questa vulnerabilità è "Input Validation and Representation".

Parte III

Implementazione e validazione delle attività di code inspection

Questa terza parte illustrerà le attività volte alla selezione delle firme di rilevamento. In riferimento a quanto scritto negli ultimi due capitoli della Parte II, il Capitolo 8 sarà dedicato alla selezione delle firme di rilevamento delle vulnerabilità relative alle "best practices" del codice C/C++, mentre il Capitolo 9 tratterà lo stesso argomento relativamente al codice Java. Nel Capitolo 10 verranno illustrate tutte le attività associate al processo di validazione ed i risultati che si sono conseguiti.

8

Selezione delle firme di rilevamento delle vulnerabilità relative alle best practices del codice C/C++

Il presente capitolo, dopo aver sottolineato la particolare importanza di alcuni aspetti espressi nei capitoli precedenti, illustrerà quali firme di rilevamento rispecchiano le "best practices" di programmazione C/C++ descritte nel Capitolo 6. Il capitolo si chiuderà evidenziando la necessità della scrittura di firme di rilevamento ad hoc.

8.1 Introduzione

Nei capitoli precedenti abbiamo visto che le attività di analisi preliminare e, in particolare, quelle di selezione delle firme di rilevamento, poggiano in primo luogo sulle relazioni introdotte nella tabella che riporta le macro-categorie di vulnerabilità indicate dalla documentazione aziendale (Tabella 4.2) e nella tabella che esprime l'associazione tra aree di esposizione e famiglie Fortify (Tabella 5.1). In tali tabelle si parte dal recepimento delle problematiche di sicurezza al fine di identificare i domini e le categorie contenenti le firme di rilevamento con una maggiore attinenza alle esigenze di code inspection del contesto di riferimento.

Le relazioni così espresse, dunque, risultano fondamentali in diversi ambiti essenziali e permettono:

- la corretta selezione delle firme, soprattutto da un punto di vista qualitativo;
- la comprensione del contesto dell'applicativo e la sua valutazione in un ambito fortemente orientato alla sicurezza;
- la migliore comprensione delle problematiche di sicurezza derivanti dalla mancata o non corretta applicazione delle linee guida per lo sviluppo di codice in sicurezza.

In base a questi presupposti, tali tabelle acquisiscono un'elevata rilevanza non più per le sole attività di analisi preliminare, ma anche per tutti i passaggi successivi, rappresentandone, così, le fondamenta.

Benché di estrema rilevanza, le tabelle di relazione costituiscono le sole fondamenta; su di esse è necessario poggiare tutti i mattoni per arrivare alla definizione della struttura definitiva. Tali mattoni sono rappresentati, essenzialmente, dalle informazioni individuate in sede di analisi avanzata per ciascuna delle problematiche segnalate; infatti, la sola presenza di una segnalazione non ne implica la pericolosità e la dannosità. In un simile contesto, diviene necessaria un'approfondita comprensione di quanto individuato.

Un esempio per meglio comprendere la problematica è costituito dalla funzionalità di "port knocking" prevista da alcuni firewall, utilizzata per abilitare una regola (tipicamente concedendo l'accesso ad uno specifico servizio) al presentarsi di uno specifico pacchetto. Dal punto di vista della sicurezza un tale comportamento può essere interpretato come una "back door" e può effettivamente esserlo se utilizzato per fini illeciti. La discriminante tra lecito e illecito può, dunque, essere individuata esclusivamente grazie alla comprensione non solo della tecnologia ma anche della logica di funzionamento dell'applicativo sottoposto a code inspection.

8.2 Selezione delle firme per il linguaggio C/C++

Il processo di code inspection individuato si articola attraverso un percorso strutturato in sette fasi, distinte e successive, all'interno delle quali vengono effettuate le operazioni che porteranno man mano a restringere il campo di osservazione agli aspetti peculiari e rilevanti dell'applicazione sotto esame e a fornire gli elementi necessari ad interpretare le vulnerabilità rilevate. Tali elementi ne consentono una valutazione oggettiva e focalizzata alle principali funzionalità applicative presenti.

Durante il processo di ispezione del codice sorgente relativo ad un dato applicativo software si procede ad un'analisi volta a definire quali aree funzionali andranno osservate in base alle peculiarità dell'applicazione sotto esame.

Sulla base delle caratteristiche dell'applicazione verranno selezionate ed evidenziate le macro-categorie, le categorie e le famiglie di vulnerabilità alle quali essa è esposta.

Una volta individuate le aree di esposizione secondo il contesto tecnologico e funzionale dell'applicazione da analizzare, si procederà con la selezione delle firme di rilevamento disponibili sullo strumento di ispezione del codice sorgente.

Le firme di rilevamento sono definizioni che identificano gli elementi nel codice sorgente che possono provocare vulnerabilità nella sicurezza o che, comunque, possono essere pericolose.

Nel caso di nostro interesse, che è un caso generico, ovvero non riferito ad un applicativo in particolare, e quindi a ben definite aree di esposizione dell'applicazione, ci si riferisce ai principi di sicurezza espressi dalle best practices. Facendo, quindi, riferimento alla Tabella 5.1 andranno selezionate le firme di rilevamento in funzione delle "best practices" trattate nei capitoli precedenti. Queste firme,

essendo diretta conseguenza delle linee guida di sviluppo in sicurezza, rivestono particolare importanza in qualsiasi processo di ispezione del codice e, per questo motivo, andranno sempre abilitate.

La Figura 8.1 mostra il procedimento "trasversale" per la selezione delle firme di rilevamento effettuata a partire dalle "best practices".

Il primo criterio in base al quale devono essere selezionate le firme di rilevamento è il linguaggio di programmazione impiegato che, nel nostro caso, è il C/C++. Infatti ogni linguaggio è caratterizzato da particolari problematiche di sicurezza e non avrebbe senso attivare il motore di analisi relativo ad un linguaggio diverso, e quindi non comprensibile, rispetto a quello impiegato.

Il software Fortify fornisce tre livelli di sicurezza, ciascuno dei quali può essere personalizzato al fine di rispettare le richieste di un'organizzazione. Per personalizzare un pacchetto di firme di rilevamento, in Fortify Audit Workbench, occorre selezionare dal menù a tendina "Tools" la voce "Manage Rulepacks". A questo punto si selezionano tutti i pacchetti di firme, oppure quello specifico del linguaggio, che, nel nostro caso, è quello relativo ai linguaggi C/C++. Successivamente si cambia il livello di sicurezza del/i rulepack in maniera opportuna (Broad, Medium o Targeted). Nel pannello in basso, si possono osservare le regole abilitate del pacchetto. La Figura 8.2 mostra la schermata principale del "Rulepack Management" in cui è selezionato il livello "Medium" di default per Fortify.

Per personalizzare un pacchetto occorre cliccare su "Customize" e selezionare le regole da abilitare in esso. Una volta effettuata la selezione, si clicca su "Apply", per applicare le nuove impostazioni, oppure su "Reset to Default", per ripristinare le impostazioni standard.

Il software Fortify Audit Workbench suddivide le firme di rilevamento in base agli analizzatori di pertinenza che sono Data Flow, Control Flow, Internal, Semantic e Structural. Ciascuno di essi gestisce una diversa tipologia di firme di rilevamento, in relazione alla tipologia di osservazione effettuata. In tal modo sarà possibile focalizzare l'analisi su specifiche tipologie di problematiche. Alcune firme, tuttavia, non sono relative ad alcun analizzatore e vengono etichettate con il termine "inputsource". Quanto scritto risulta evidente dalla Figura 8.3 in cui è mostrata la finestra di gestione delle firme di rilevamento dei linguaggi C/C++. La struttura della finestra dopo la selezione delle firme di nostro interesse è riportata nell'Appendice A. In essa non è stata selezionata ancora alcuna firma.

8.2.1 Input Source

Relativamente alle firme etichettate "inputsource" dovrà essere abilitata la seguente firma di rilevamento:

- *File System.* La selezione di questa firma si riferisce al fatto che un buffer di grandezza inaspettata da parte di una funzione di manipolazione del percorso potrebbe causare un buffer overflow.

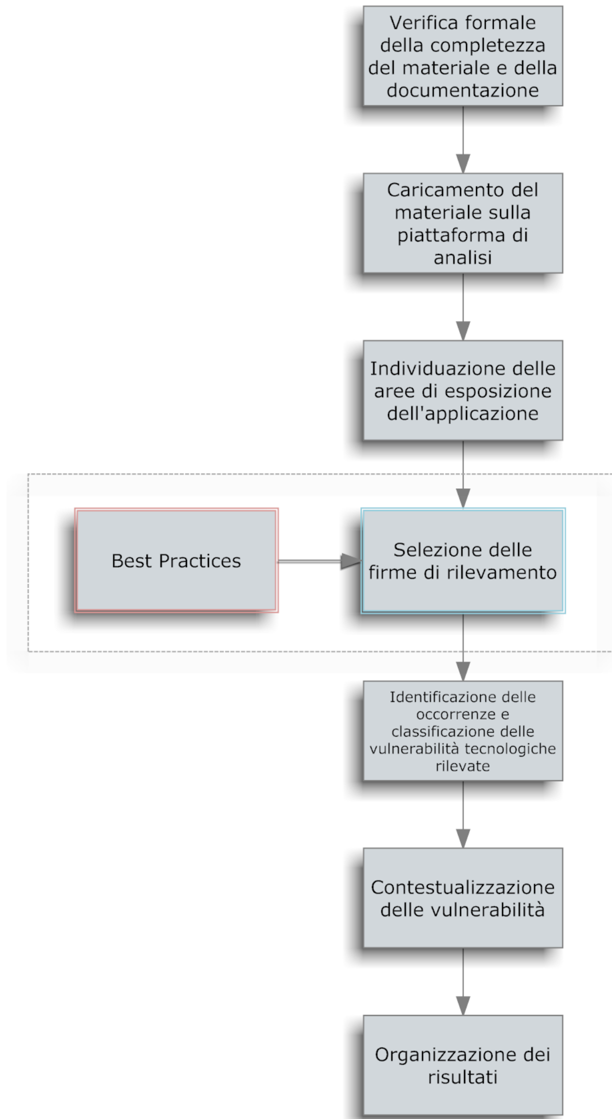


Figura 8.1. Selezione delle firme di rilevamento a partire dalle best practices

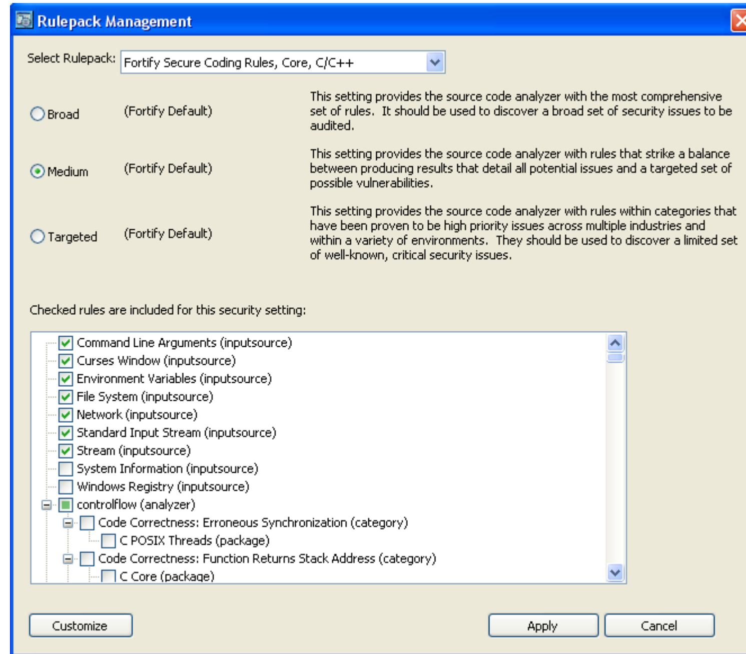


Figura 8.2. Rulepack Management: Schermata principale

La Figura 8.4 riporta la selezione delle firme etichettate "inputsource" che rispecchia le direttive delle "best practices".

8.2.2 Control Flow

L'analizzatore Control Flow consente la rilevazione dei concatenamenti di funzioni e processi interni all'applicativo che potrebbero portare al verificarsi di problematiche di sicurezza.

Esso utilizza firme per fornire le definizioni degli stati della macchina che hanno un comportamento pericoloso e le applica al codice sorgente per rilevare le vulnerabilità; in questo modo è in grado di individuare sequenze di operazioni potenzialmente dannose in una singola funzione o metodo.

Tramite l'analisi dei percorsi del flusso del programma, l'analizzatore Control Flow può determinare se un insieme di operazioni vengono eseguite in un ordine che viola un vincolo temporale di sicurezza.

In aggiunta all'identificazione di sequenze di chiamate di funzione pericolose, l'analizzatore Control Flow può rilevare l'assenza di una chiamata di funzione necessaria da parte di una specifica sequenza.

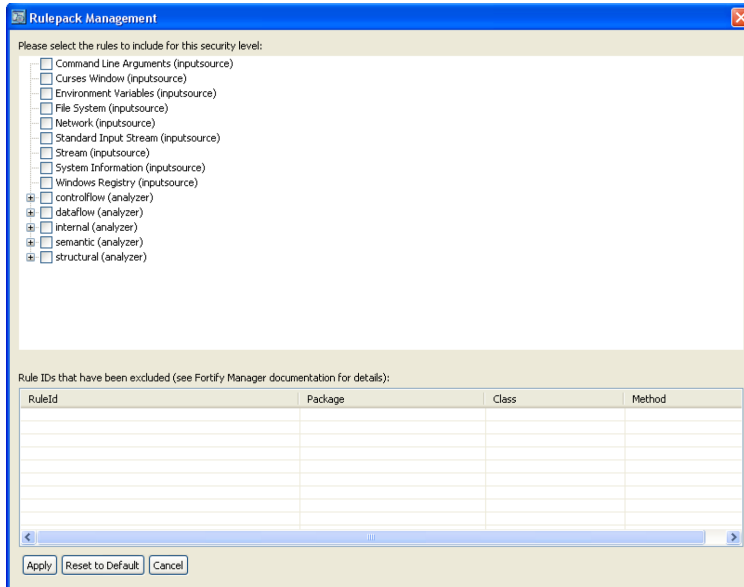


Figura 8.3. Rulepack Management: Selezione firme

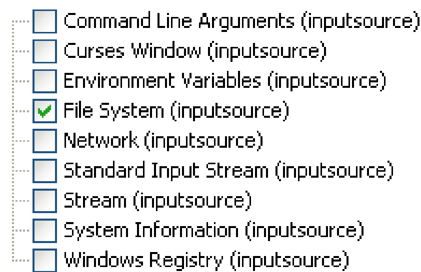


Figura 8.4. Selezione delle firme di rilevamento relative a "inputsource"

Esempi di utilizzo dell'analizzatore Control Flow sono l'individuazione delle vulnerabilità "time of check/time of use", l'identificazione dell'utilizzo di variabili prima che siano inizializzate ed il controllo volto ad assicurare che una utility, come un parser XML, venga configurata correttamente prima di essere utilizzata.

Relativamente all'analizzatore Control Flow dovranno essere abilitate le seguenti firme di rilevamento:

- *Double Free*. La selezione di questa firma si riferisce al fatto che un buffer overflow potrebbe essere causato da una doppia chiamata `free()` sullo stesso indirizzo di memoria.

- *Insecure Temporary File*. La selezione di questa firma si riferisce al fatto che ogni nome di file temporaneo deve essere unico e non predicibile.
- *Memory Leak*. La selezione di questa firma si riferisce al fatto che l'applicazione deve provvedere all'allocazione ed alla deallocazione della propria memoria.
- *Missing Check against Null*. La selezione di questa firma si riferisce al fatto che il tipo NULL deve essere corretto mediante casting quando viene passato come parametro ad una funzione.
- *Null Dereference*. La selezione di questa firma si riferisce al fatto che occorre gestire opportunamente i puntatori a NULL.
- *String Termination Error*. La selezione di questa firma si riferisce al fatto che tutte le stringhe devono essere terminate dal carattere NULL.
- *Uninitialized Variable*. La selezione di questa firma si riferisce al fatto che tutte le variabili locali devono essere inizializzate prima di essere utilizzate.
- *Use After Free*. La selezione di questa firma si riferisce al fatto che non devono esistere puntatori a risorse distrutte.

La Figura 8.5 riporta la selezione delle firme relative all'analizzatore Control Flow che rispecchia le direttive delle "best practices".

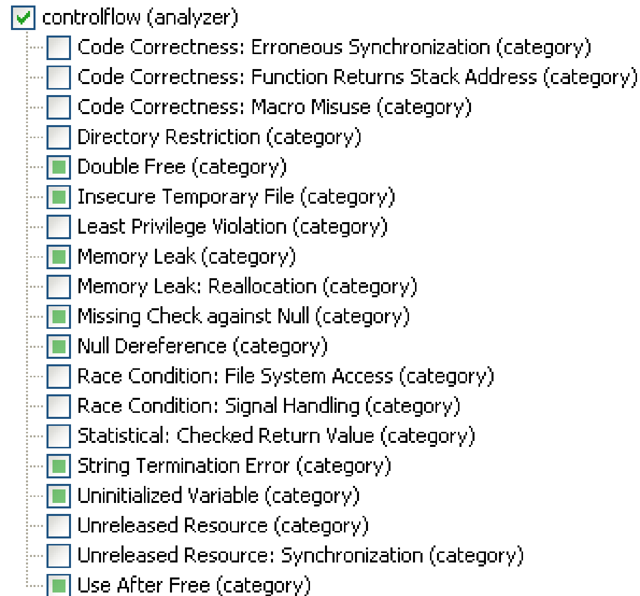


Figura 8.5. Selezione delle firme di rilevamento relative all'analizzatore Control Flow

8.2.3 Data Flow

L'analizzatore Data Flow consente la rilevazione delle criticità legate principalmente al flusso dei dati, con particolare attenzione ai punti di ingresso che coinvolgono gli utenti. Esso traccia l'input utente a partire dal suo ingresso al programma seguendone la propagazione attraverso le funzioni e le variabili; le firme relative all'analizzatore Data Flow identificano funzioni che possono assumere diversi ruoli e permettono a Fortify SCA di realizzare un'accurata analisi del flusso dei dati.

L'analizzatore Data Flow utilizza segni di tracciamento globali e intra-procedurali per rilevare il flusso dei dati tra la sorgente (il luogo di inserimento dati) e la destinazione (la chiamata di funzione o l'operazione pericolose).

Il comportamento di una firma di data flow spesso si basa su argomenti in entrata e in uscita di una funzione; le firme descrivono il flusso dei dati e sono una rappresentazione logica dei dati prodotti o distrutti da una specifica funzione. Esempi includono l'input utente costituito da una stringa di lunghezza illimitata copiata all'interno di un buffer dimensionato staticamente, oppure l'input utente costituito da una stringa utilizzata per la costruzione di una query SQL.

Relativamente all'analizzatore Data Flow dovranno essere abilitate le seguenti firme di rilevamento:

- *Buffer Overflow*. La selezione di questa firma si riferisce al fatto che tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati.
- *Command Injection (Data Flow)*. La selezione di questa firma si riferisce al fatto che l'input proveniente dall'utente deve sempre essere convalidato e scremato da caratteri non validi (“;” “|” “!” “&” “~” “,” “"” “-” “*” “%” “'” “\” “/” “<” “>” “?” “$” “@” “:” “(” “)” “[” “]” “{” “}” “.”) prima di essere passato alle successive elaborazioni dell'applicazione (ad esempio, alla funzione `system()`).
- *Format String*. La selezione di questa firma si riferisce al fatto che un buffer overflow può essere causato da un mancato controllo della formattazione delle stringhe da parte delle funzioni; ad esempio, il codice non deve tentare di operare su una stringa (o un array di caratteri) che non è terminato dal carattere NULL.
- *Integer Overflow*. La selezione di questa firma si riferisce al fatto che tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati; un buffer overflow può essere causato da un integer overflow.
- *String Termination Error*. La selezione di questa firma si riferisce al fatto che tutte le stringhe devono essere terminate dal carattere NULL.
- *Unsafe Reflection*. La selezione di questa firma si riferisce al fatto che un utilizzo scorretto delle variabili di controllo, delle istruzioni switch, del passaggio degli argomenti, dei valori di ritorno, delle chiamate di funzione e dei file potrebbe modificare il percorso del flusso di controllo dell'applicazione aggirando i controlli di sicurezza.

La Figura 8.6 riporta la selezione delle firme relative al flusso dei dati che rispecchia le direttive delle “best practices”.

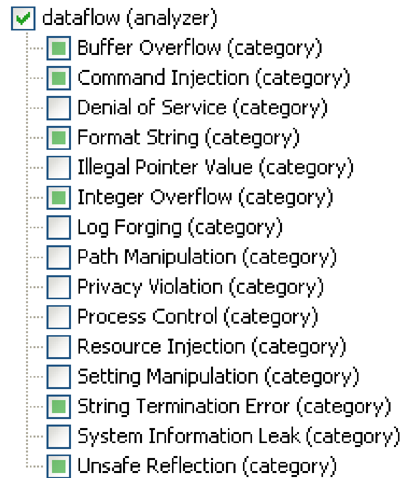


Figura 8.6. Selezione delle firme di rilevamento relative all’analizzatore Data Flow

8.2.4 Internal

Relativamente all’analizzatore Internal dovranno essere abilitate le seguenti firme di rilevamento:

- *Buffer Overflow*. La selezione di questa firma si riferisce al fatto che tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati.
- *Format String: Argument Type/Number Mismatch*. La selezione di questa firma si riferisce al fatto che l’utilizzo scorretto degli argomenti potrebbe provocare errori di casting e problematiche di gestione delle variabili numeriche.

La Figura 8.7 riporta la selezione delle firme relative all’analizzatore Internal che rispecchia le direttive delle “best practices”.

8.2.5 Semantic

L’analizzatore semantico consente la rilevazione di utilizzi di funzioni e/o API a livello intra-procedurale potenzialmente critici. Esso ha una logica specializzata per la rilevazione di buffer overflow, format string e vulnerabilità di tipo execution

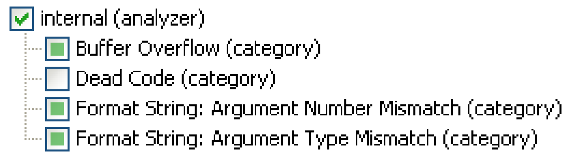


Figura 8.7. Selezione delle firme di rilevamento relative all'analizzatore Internal

path, in aggiunta ad altri tipi di vulnerabilità che dipendono da un'analisi più generale.

Per esempio, ogni utilizzo di funzioni potenzialmente dannose può essere segnalato dall'analizzatore semantico che le identifica in modo intelligente tramite dettagli significativi.

Relativamente all'analizzatore semantico dovranno essere abilitate le seguenti firme di rilevamento:

- *Buffer Overflow*. La selezione di questa firma si riferisce al fatto che tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati.
- *Command Injection (Semantic)*. La selezione di questa firma si riferisce al fatto che l'input proveniente dall'utente deve sempre essere convalidato e scremato da caratteri non validi prima di essere passato alle successive elaborazioni dell'applicazione.
- *Format String*. La selezione di questa firma si riferisce al fatto che un buffer overflow può essere causato da un mancato controllo della formattazione delle stringhe da parte delle funzioni.
- *Heap Inspection*. La selezione di questa firma si riferisce al fatto che, in un sorgente C++, è meglio utilizzare `new` invece di `realloc()` pur essendo quest'ultima una funzione standard del C.
- *Insecure Compiler Optimization*. La selezione di questa firma si riferisce al fatto che i risultati dei controlli e delle procedure di sicurezza, nonché i relativi dati, non devono risiedere in memoria per lunghi periodi.
- *Insecure Temporary File*. La selezione di questa firma si riferisce al fatto che ogni nome di file temporaneo deve essere unico e non predicibile.
- *Often Misused: Strings*. La selezione di questa firma si riferisce al fatto che un buffer overflow può essere causato da funzioni che convertono tra stringhe Multibyte e Unicode.
- *Unchecked Return Value*. La selezione di questa firma si riferisce al fatto che i valori di ritorno di tutte le chiamate di sistema devono essere controllati per determinare lo stato di esecuzione del programma.

La Figura 8.8 riporta la selezione delle firme relative all'analizzatore semantico che rispecchia le direttive delle "best practices".

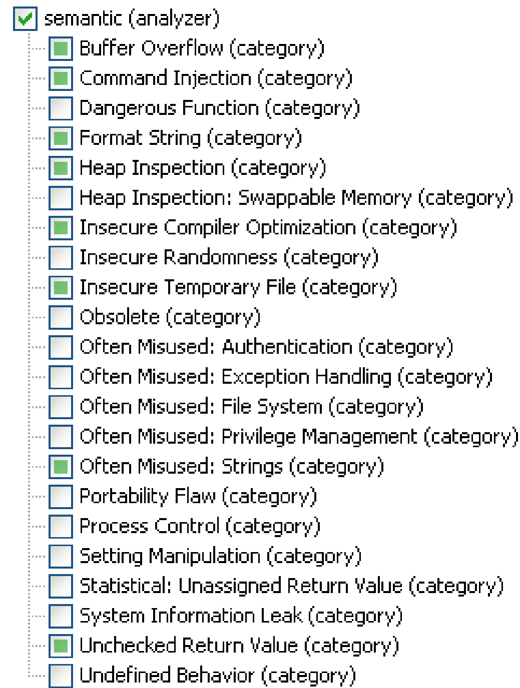


Figura 8.8. Selezione delle firme di rilevamento relative all'analizzatore Semantic

8.2.6 Structural

L'analizzatore strutturale consente la rilevazione delle criticità presenti in un programma a livello di struttura.

Relativamente all'analizzatore strutturale dovranno essere abilitate le seguenti firme di rilevamento:

- *Type Mismatch: Negative to Unsigned*. La selezione di questa firma si riferisce al fatto che possono scaturire errori di casting e problematiche di gestione delle variabili numeriche se la funzione restituisce un valore negativo invece di uno `unsigned`.
- *Type Mismatch: Signed to Unsigned*. La selezione di questa firma si riferisce al fatto che possono scaturire errori di casting e problematiche di gestione delle variabili numeriche se la funzione restituisce un valore `signed` invece di uno `unsigned`.

La Figura 8.9 riporta la selezione delle firme relative all'analizzatore strutturale che rispecchia le direttive delle "best practices".

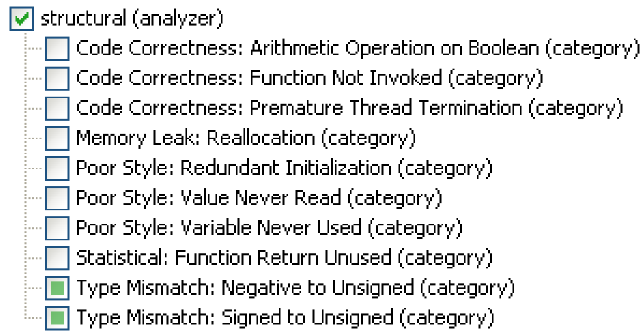


Figura 8.9. Selezione delle firme di rilevamento relative all'analizzatore Structural

8.3 Firme ad hoc

Nelle loro varie forme, le firme di rilevamento racchiudono la conoscenza che permette di individuare quali dati siano potenzialmente pericolosi all'interno del flusso del programma, quali funzioni siano intrinsecamente insicure, oppure in quale contesto o specifica sequenza di programma lo diventino. I pacchetti di regole messi a disposizione da Secure Coding Rulepacks rilevano migliaia di costrutti di codice vulnerabili e i possibili pericoli nell'utilizzo dei dati.

Ciò nonostante, non tutte le "best practices" trovano una firma corrispondente tra i pacchetti messi a disposizione dello strumento di analisi utilizzato; si rende, quindi, necessario estendere le funzionalità di Fortify SCA o di Secure Coding Rulepacks creando delle firme di rilevamento personalizzate tramite Rules Builder.

Quest'ultimo trova largo impiego in tutti i casi in cui si devono rispettare le stringenti linee guida di sicurezza di un'organizzazione oppure si deve analizzare un progetto che utilizza librerie o codici binari pre-compilati di terze parti che non sono compresi in Secure Coding Rulepacks.

Da notare, per ultimo, che il Rules Buider permette la modifica di tutti i tipi di firme eccetto quelle strutturali.

9

Selezione delle firme di rilevamento delle vulnerabilità relative alle best practices del codice Java

Il presente capitolo, dopo una breve introduzione, illustrerà, in analogia con il capitolo precedente, quali firme di rilevamento rispecchiano le "best practices" di programmazione Java descritte nel Capitolo 7. Anche questo capitolo si chiuderà sottolineando la necessità della scrittura di firme di rilevamento ad hoc.

9.1 Introduzione

Nel precedente capitolo abbiamo illustrato la selezione delle firme di rilevamento relative ai linguaggi C/C++ a partire dalle relative best practices. Un procedimento analogo verrà adesso esposto relativamente al linguaggio Java. Ai fini della selezione si terrà bene in conto il fatto che il 90% delle vulnerabilità trovate nel software è da attribuirsi a due distinte macro-categorie di errori di programmazione, ovvero a una poco accorta gestione dell'input utente e a controlli erronei o assolutamente assenti durante l'allocazione delle aree di memoria adibite a contenere i dati.

Verranno considerate anche le best practices presentate nel Capitolo 6 che, sebbene riferite ai linguaggi C/C++, costituiscono delle pratiche di programmazione che, sotto alcuni aspetti, possono essere considerate di carattere più generale.

Anche in questo caso, al fine di identificare i domini e le categorie contenenti le firme di rilevamento più attinenti alle esigenze di code inspection del contesto di riferimento, rivestiranno particolare importanza le relazioni tra le aree di esposizione e le famiglie Fortify espresse nella Tabella 5.1.

9.2 Selezione delle firme per il linguaggio Java

Nel caso di nostro interesse, che è un caso generico, ovvero non riferito ad un applicativo in particolare, e quindi a ben definite aree di esposizione dell'applica-

zione, ci si riferisce ai principi di sicurezza espressi dalle best practices. Facendo, quindi, riferimento alla Tabella 5.1 andranno selezionate le firme di rilevamento in funzione delle "best practices" trattate nei capitoli precedenti. Queste firme, essendo diretta conseguenza delle linee guida di sviluppo in sicurezza, rivestono particolare importanza in qualsiasi processo di ispezione del codice e, per questo motivo, andranno sempre abilitate.

Il procedimento "trasversale" per la selezione delle firme di rilevamento, effettuata a partire dalle "best practices", è sempre quello mostrato in figura Figura 8.1.

Come già detto, il primo criterio in base al quale devono essere selezionate le firme di rilevamento è il linguaggio di programmazione impiegato che, in questo caso, è Java. Infatti ogni linguaggio è caratterizzato da particolari problematiche di sicurezza e non avrebbe senso attivare il motore di analisi relativo ad un linguaggio diverso (e quindi non comprensibile) rispetto a quello correntemente utilizzato.

Il software Fortify fornisce tre livelli di sicurezza, ciascuno dei quali può essere personalizzato al fine di rispettare le richieste di un'organizzazione. Per personalizzare un pacchetto di firme di rilevamento, in Fortify Audit Workbench, occorre selezionare dal menù a tendina "Tools" la voce "Manage Rulepacks". A questo punto si selezionano tutti i pacchetti di firme, oppure quello specifico del linguaggio, che, nel nostro caso, è Java. Successivamente si cambia il livello di sicurezza del/i rulepack in maniera opportuna (Broad, Medium o Targeted). Nel pannello in basso si possono osservare le regole del pacchetto abilitate. La Figura 9.1 mostra la schermata principale del "Rulepack Management" in cui è selezionato il livello "Medium" di default per Fortify.

Per personalizzare un pacchetto occorre cliccare su "Customize" e selezionare le regole da abilitare in esso. Una volta effettuata la selezione, si clicca su "Apply", per applicare le nuove impostazioni, oppure su "Reset to Default", per ripristinare quelle standard.

Il software Fortify Audit Workbench suddivide le firme di rilevamento in base agli analizzatori di pertinenza che sono Data Flow, Control Flow, Internal, Semantic e Structural. Ciascuno di essi gestisce una diversa tipologia di firme di rilevamento, in relazione alla tipologia di osservazione effettuata. In tal modo sarà possibile focalizzare l'analisi su specifiche tipologie di problematiche. Alcune firme, tuttavia, non sono relative ad alcun analizzatore e vengono etichettate con il termine "inputsource".

La Figura 9.1 mostra la schermata principale del "Rulepack Management" in cui è selezionato il livello "Medium" di default per Fortify mentre la Figura 9.2 mostra la finestra di gestione delle firme di rilevamento del linguaggio Java. La struttura della finestra dopo la selezione delle firme di nostro interesse è riportata nell'Appendice B. In essa non è stata selezionata ancora alcuna firma.

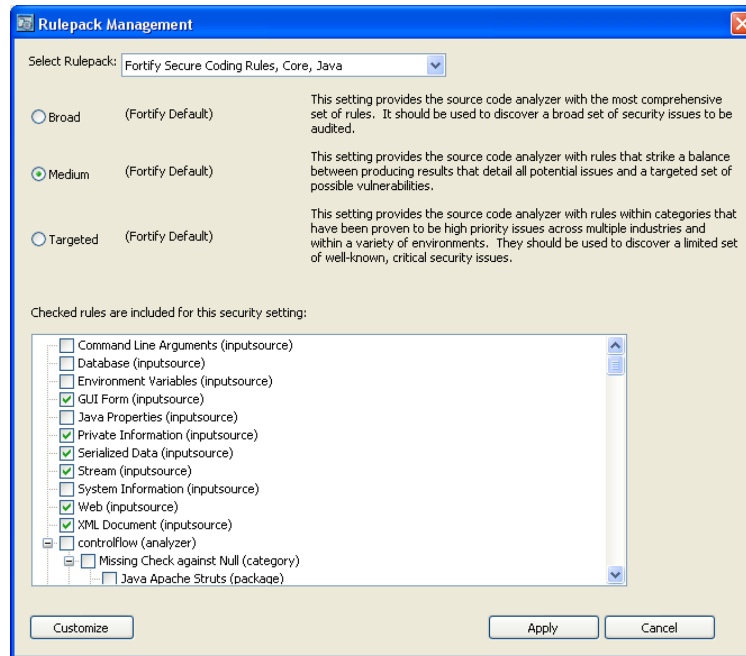


Figura 9.1. Rulepack Management: Schermata principale

Nel caso relativo al linguaggio Java si nota che alcune pratiche di programmazione sicura fanno riferimento a delle vulnerabilità che non trovano riscontro nelle firme che il software Fortify indica con il nome di "Fortify Secure Coding Rules, Core, Java". Per questo motivo occorrerà fare riferimento al pacchetto di firme "Fortify Secure Coding Rules, Extended, Java". Esso gestisce la sicurezza relativa alle API di varie librerie estese e di terze parti includendo MS, JNDI, J2EE, Apache Commons, Log4J, ORO, Struts, ATG Dynamo, Hibernate, Spring e iBatis. In Figura 9.3 viene indicata la selezione di questo pacchetto.

Per non appesantire la notazione le firme relative a questo pacchetto "esteso" verrano marcate, nelle figure successive, con un quadratino rosso anziché verde.

9.2.1 Input Source

Relativamente alle firme etichettate "inputsource" dovranno essere abilitate le seguenti firme di rilevamento:

- *Private Information.* La selezione di questa firma si riferisce al fatto che le informazioni riservate, come chiavi crittografiche, password e certificati, non devono mai essere inserite e presenti all'interno del codice o nelle librerie utilizzate.

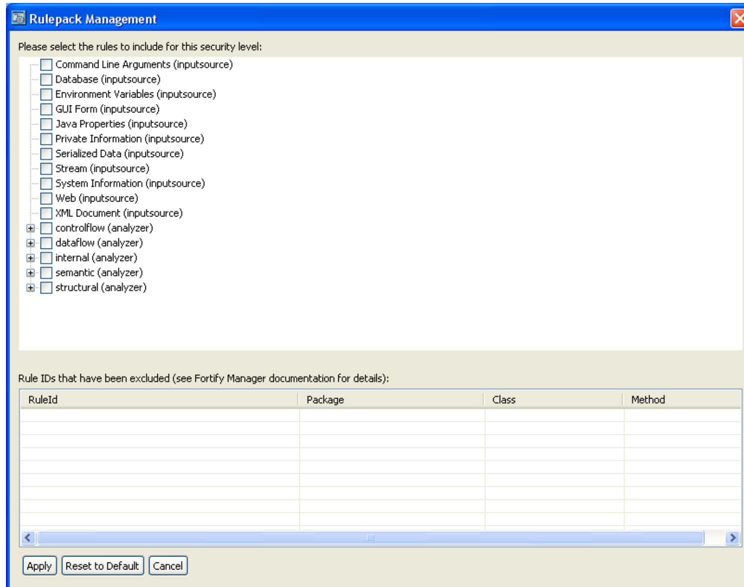


Figura 9.2. Rulepack Management: Selezione firme

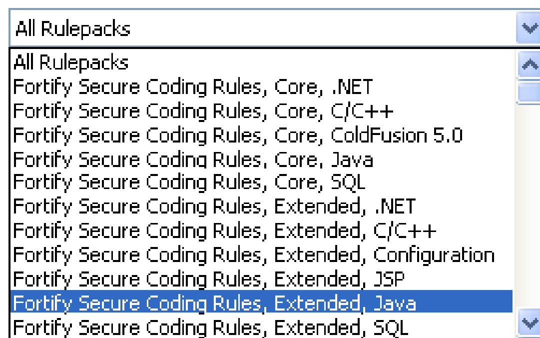


Figura 9.3. Selezione del pacchetto di firme “Fortify Secure Coding Rules, Extended, Java”

- *Serialized Data.* La selezione di questa firma si riferisce al fatto che classi ed oggetti non dovrebbero mai essere serializzabili.
- *Web.* La selezione di questa firma si riferisce al fatto che tutte le eccezioni che si verificano durante l’esecuzione delle Servlet che costituiscono l’applicazione Web devono essere catturate e gestite opportunamente. I relativi messaggi di errore sollevati dovrebbero essere visualizzati secondo uno schema dettagliato. Il meccanismo di gestione degli errori dovrebbe essere in grado di gestire ogni

tipo di dati in ingresso e, nel contempo, dovrebbe garantire la sicurezza. Dovrebbero essere previsti dei messaggi di errore semplici, in grado di indicare la causa e di archiviare i tentativi di intrusione in modo tale da poterli verificare in un secondo tempo. Inoltre, la gestione degli errori non dovrebbe essere concentrata soltanto sui dati forniti in ingresso dall'utente, ma dovrebbe anche includere tutti gli errori che possono essere generati da componenti interni come system call, query sul database o altre funzioni interne.

La Figura 9.4 riporta la selezione delle firme etichettate "inputsource" che rispecchia le direttive delle "best practices".

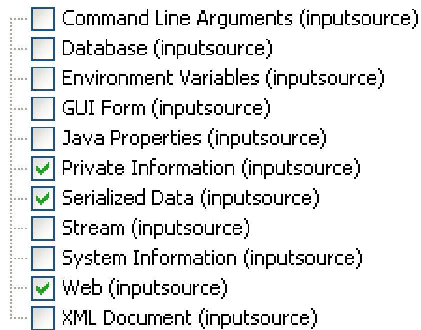


Figura 9.4. Selezione delle firme di rilevamento relative a "inputsource"

9.2.2 Control Flow

L'analizzatore Control Flow consente la rilevazione dei concatenamenti di funzioni e processi interni all'applicativo che potrebbero portare al verificarsi di problematiche di sicurezza.

Esso utilizza firme per fornire le definizioni degli stati della macchina che hanno un comportamento pericoloso e le applica al codice sorgente per rilevare le vulnerabilità; in questo modo è in grado di individuare sequenze di operazioni potenzialmente dannose in una singola funzione o metodo.

Tramite l'analisi dei percorsi del flusso del programma, l'analizzatore Control Flow può determinare se un insieme di operazioni vengono eseguite in un ordine che viola un vincolo temporale di sicurezza.

In aggiunta all'identificazione di sequenze di chiamate di funzione pericolose, l'analizzatore Control Flow può rilevare l'assenza di una chiamata di funzione necessaria da parte di una specifica sequenza.

Esempi di utilizzo dell'analizzatore Control Flow sono l'individuazione delle vulnerabilità "time of check/time of use", l'identificazione dell'utilizzo di variabili

prima che vengano inizializzate ed il controllo volto ad assicurare che una utility, come un parser XML, venga configurata correttamente prima di essere utilizzata.

Relativamente all'analizzatore Control Flow andranno abilitate le seguenti firme di rilevamento:

- *Missing Check against Null*. La selezione di questa firma si riferisce al fatto che occorre prestare la dovuta attenzione al tipo NULL quando viene passato come parametro ad una funzione. Nel caso in cui un input nullo crei un'eccezione, il programma restituirà un errore sconosciuto.
- *Null Dereference*. La selezione di questa firma si riferisce al fatto che occorre gestire opportunamente i puntatori a NULL.
- *Session Fixation*. La selezione di questa firma si riferisce al fatto che non ci si deve mai basare sulla presenza o meno di un cookie o di un token di sessione precedentemente memorizzato per autenticare un utente; infatti, ogni identificatore di sessione dovrebbe essere invalidato prima di procedere ad una nuova autenticazione. Se così non fosse un attaccante avrebbe l'opportunità di appropriarsi di sessioni autenticate impersonificando un altro utente.

La Figura 9.5 riporta la selezione delle firme relative all'analizzatore Control Flow che rispecchia le direttive delle "best practices".

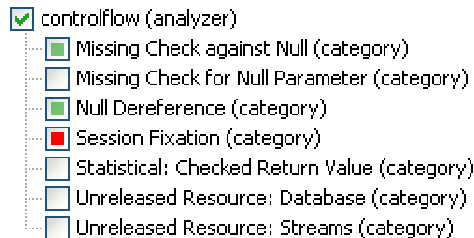


Figura 9.5. Selezione delle firme di rilevamento relative all'analizzatore Control Flow

9.2.3 Data Flow

L'analizzatore Data Flow consente la rilevazione delle criticità legate principalmente al flusso dei dati, con particolare attenzione ai punti di ingresso che coinvolgono gli utenti. Esso traccia l'input utente a partire dal suo ingresso al programma seguendone la propagazione attraverso le funzioni e le variabili; le firme relative all'analizzatore Data Flow identificano funzioni che possono assumere diversi ruoli e permettono a Fortify SCA di realizzare un'accurata analisi del flusso dei dati.

L'analizzatore Data Flow utilizza segni di tracciamento globali e intra-procedurali per rilevare il flusso dei dati tra la sorgente, ovvero il luogo di inserimento dati, e la destinazione, ovvero la chiamata di funzione o l'operazione pericolose.

Il comportamento di una firma di data flow spesso dipende dagli argomenti in entrata e in uscita di una funzione; le firme descrivono il flusso dei dati e sono una rappresentazione logica dei dati prodotti o distrutti da una specifica funzione. Esempi includono l'input utente costituito da una stringa di lunghezza illimitata copiata all'interno di un buffer dimensionato staticamente, oppure l'input utente costituito da una stringa utilizzata per la costruzione di una query SQL.

Relativamente all'analizzatore Data Flow dovranno essere abilitate le seguenti firme di rilevamento:

- *Code Correctness: Erroneous Class Compare*. La selezione di questa firma si riferisce al fatto che è necessario evitare il confronto degli oggetti basato sul loro nome. Se così non fosse si potrebbe provocare un comportamento inaspettato oppure consentire ad un attaccante l'introduzione di una classe malevola.
- *Command Injection (Data Flow)*. La selezione di questa firma si riferisce al fatto che l'input proveniente dall'utente deve sempre essere convalidato e scremato da caratteri non validi (“;” “|” “!” “&” “~” “,” “"” “-” “*” “%” “'” “\” “/” “<” “>” “?” “$” “@” “:” “(” “)” “[” “]” “{” “}” “.”) prima di essere passato alle successive elaborazioni dell'applicazione (ad esempio, `System.exec()`).
- *Cross-Site Scripting*. La selezione di questa firma si riferisce al fatto che l'invio di dati non validati ad un browser Web potrebbe portare all'esecuzione di codice malevolo. Questa vulnerabilità è, per così dire, tra le più attuali.
- *Denial of Service*. La selezione di questa firma si riferisce al fatto che è opportuno implementare meccanismi che consentono di limitare al massimo il numero di risorse allocate per ogni singolo utente; in altre parole, è necessario stabilire una quota per ogni utente, soprattutto se non autenticato, in modo tale da poter limitare il carico massimo che un utente può applicare al sistema. In caso contrario, un malintenzionato potrebbe provocare il blocco di un programma oppure renderlo non disponibile agli altri utenti autorizzati.
- *HTTP Response Splitting*. La selezione di questa firma si riferisce al fatto che, in tutti i casi in cui sia possibile, è opportuno verificare il campo “Referer” dell'intestazione HTTP e rigettare le informazioni provenienti da host o link incorretti e/o inaspettati. Infatti, l'inserimento di dati HTTP non validati in un header di risposta può causare cache-poisoning, cross-site scripting e attacchi di tipo page hijacking.
- *Privacy Violation*. La selezione di questa firma si riferisce al fatto che tutte le transazioni che prevedono la ricezione, da parte di una Servlet, di dati personali e/o sensibili (ad esempio, login, password, fede religiosa, etc.) devono sempre avvenire in modo sicuro e, almeno, su protocollo HTTP protetto tramite SSL (HTTPS). In quest'ottica, il processo di autenticazione e di autorizzazione di un utente deve sempre avvenire tramite protocollo HTTPS. Un trattamento scorretto di informazioni sensibili potrebbe compromettere la privacy degli utenti ed è spesso illegale.

- *SQL Injection.* La selezione di questa firma si riferisce al fatto che se una Servlet accede ad un database, i relativi comandi SQL non devono mai essere costruiti utilizzando concatenazioni di stringhe, né, tanto meno, concatenazioni di informazioni inserite dagli utenti, anche se verificate e validate. Consentire ad un utente la costruzione di un'espressione dinamica SQL potrebbe permettere ad un attaccante di modificare opportunamente l'espressione per eseguire comandi arbitrari.
- *Unsafe Reflection.* La selezione di questa firma si riferisce al fatto che un utilizzo scorretto delle variabili di controllo, delle istruzioni switch, del passaggio degli argomenti, dei valori di ritorno, delle chiamate di funzione e dei file potrebbe modificare il percorso del flusso di controllo dell'applicazione aggirando i controlli di sicurezza.

La Figura 9.6 riporta la selezione delle firme relative al flusso dei dati che rispecchia le direttive delle "best practices".

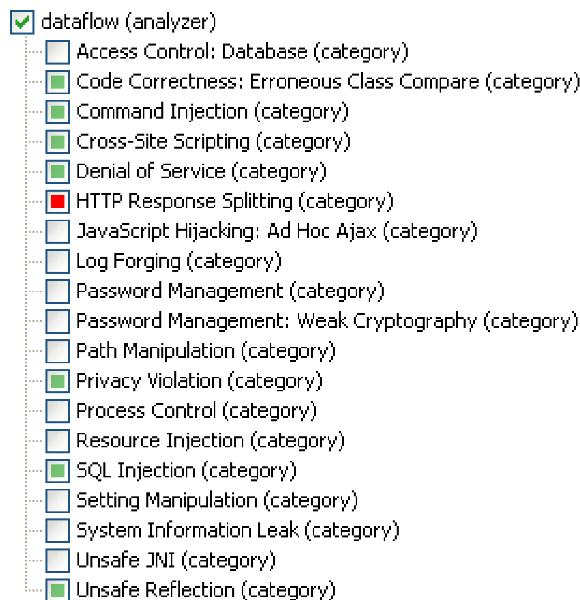


Figura 9.6. Selezione delle firme di rilevamento relative all'analizzatore Data Flow

9.2.4 Internal

Relativamente all'analizzatore Internal dovrà essere abilitata la seguente firma di rilevamento:

- *Session Fixation.* La selezione di questa firma si riferisce al fatto che non ci si deve mai basare sulla presenza o meno di un cookie o di un token di sessione precedentemente memorizzato per autenticare un utente. Infatti, ciascun identificatore di sessione dovrebbe essere invalidato prima di procedere ad una nuova autenticazione. Se così non fosse un attaccante avrebbe l'opportunità di appropriarsi di sessioni autenticate impersonificando un altro utente.

La Figura 9.7 riporta la selezione delle firme relative all'analizzatore Internal che rispecchia le direttive delle "best practices".

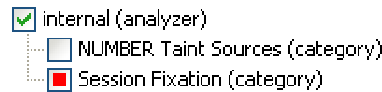


Figura 9.7. Selezione delle firme di rilevamento relative all'analizzatore Internal

9.2.5 Semantic

L'analizzatore semantico consente la rilevazione di utilizzi di funzioni e/o API a livello intra-procedurale potenzialmente critici. Esso ha una logica specializzata per la rilevazione di buffer overflow, format string e vulnerabilità di tipo execution path, in aggiunta ad altri tipi di vulnerabilità che dipendono da un'analisi più generale.

Ad esempio, ogni utilizzo di funzioni potenzialmente dannose può essere segnalato dall'analizzatore semantico che le identifica in modo intelligente tramite dettagli significativi. Relativamente all'analizzatore semantico dovranno essere abilitate le seguenti firme di rilevamento:

- *Command Injection (Semantic).* La selezione di questa firma si riferisce al fatto che l'input proveniente dall'utente deve sempre essere convalidato e scremato dai caratteri non validi prima di essere passato alle successive elaborazioni dell'applicazione.
- *Denial of Service.* La selezione di questa firma si riferisce al fatto che è opportuno implementare meccanismi che consentono di limitare al massimo il numero di risorse allocate per ogni singolo utente. In altre parole, è necessario stabilire una quota per ogni utente, soprattutto se non autenticato, in modo da poter limitare il carico massimo che egli può applicare al sistema; qualora ciò non dovesse avvenire, un malintenzionato potrebbe provocare il blocco di un programma oppure renderlo non disponibile agli altri utenti autorizzati.
- *SQL Injection.* La selezione di questa firma si riferisce al fatto che se una Servlet accede ad un database, i relativi comandi SQL non devono mai essere costruiti utilizzando concatenazioni di stringhe, né tanto meno concatenazioni

di informazioni inserite dagli utenti, anche se verificate e validate. Consentire ad un utente la costruzione di un'espressione dinamica SQL potrebbe permettere ad un attaccante di modificare opportunamente l'espressione per eseguire comandi arbitrari.
- *Unchecked Return Value*. La selezione di questa firma si riferisce al fatto che i valori di ritorno di tutte le chiamate di sistema devono essere controllati per determinare lo stato di esecuzione del programma.

La Figura 9.8 riporta la selezione delle firme relative all'analizzatore semantico che rispecchia le direttive delle "best practices".

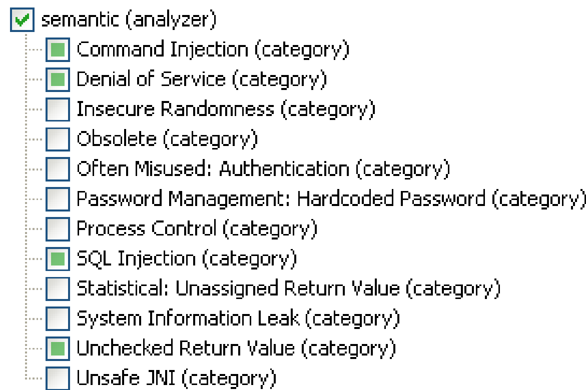


Figura 9.8. Selezione delle firme di rilevamento relative all'analizzatore semantico

9.2.6 Structural

L'analizzatore strutturale consente la rilevazione delle criticità, a livello di struttura, presenti in un programma.

Relativamente all'analizzatore strutturale dovranno essere abilitate le seguenti firme di rilevamento:

- *Code Correctness: Erroneous Class Compare*. La selezione di questa firma si riferisce al fatto che è necessario evitare il confronto degli oggetti in base al loro nome. Se così non fosse, si potrebbe provocare un comportamento inaspettato oppure si potrebbe consentire ad un attaccante l'introduzione di una classe malevola.
- *Code Correctness: Misspelled Method Name*. La selezione di questa firma si riferisce al fatto che non si deve mai effettuare l'overriding dei metodi, quali `wait()` e `notify()`, della classe `java.lang.Thread`. Infatti, la sovrascrittura di un metodo comune Java probabilmente non ha l'effetto desiderato.

- *Unsafe Mobile Code: Inner Class*. La selezione di questa firma si riferisce al fatto che l'utilizzo di "Inner Class" dovrebbe essere evitato sempre e, quindi, anche nell'ambito del codice "mobile".
- *Unsafe Mobile Code: Unsafe Array Declaration*. La selezione di questa firma si riferisce al fatto che l'utilizzo di variabili di tipo `static` dovrebbe essere evitato per quanto possibile e, quindi, anche nell'ambito del codice "mobile".
- *Unsafe Mobile Code: Unsafe Public Field*. La selezione di questa firma si riferisce al fatto che le classi, i metodi e le variabili dovrebbero essere definiti come `final` sempre e, quindi, anche nell'ambito del codice "mobile".

La Figura 9.9 riporta la selezione delle firme relative all'analizzatore strutturale che rispecchia le direttive delle "best practices".

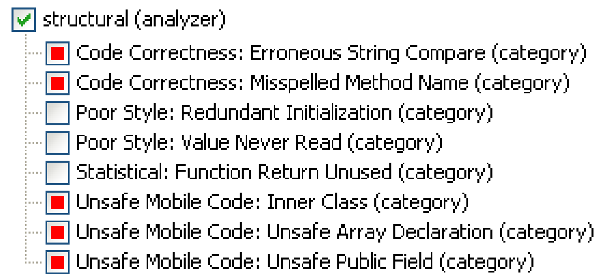


Figura 9.9. Selezione delle firme di rilevamento relative all'analizzatore strutturale

9.3 Firme ad hoc

Nonostante il Secure Coding Rulepacks permetta di rilevare migliaia di costrutti di codice vulnerabili e i possibili pericoli nell'utilizzo dei dati, anche in questo caso, non tutte le "best practices" trovano una firma corrispondente tra i pacchetti messi a disposizione dello strumento di analisi utilizzato; si rende, quindi, necessario estendere le funzionalità di Fortify SCA o di Secure Coding Rulepacks creando delle firme di rilevamento personalizzate tramite Rules Builder.

Ricordiamo che quest'ultimo permette la modifica di tutti i tipi di firme eccetto quelle strutturali.

10

Validazione della selezione delle firme di rilevamento

Il presente capitolo illustrerà l'applicazione delle firme di rilevamento, descritte nel Capitolo 9, scelte in base alle "best practices" di programmazione Java. Dapprima verranno effettuate alcune considerazioni matematiche sulla fase di selezione delle firme di rilevamento. Successivamente verrà sottoposto al processo di Code Inspection un codice sorgente inerente la nostra realtà di riferimento. Tale codice, insieme ad un altro, sarà utilizzato al fine di verificare quanto la selezione di firme da noi considerata sia consona alle finalità per le quali è stata prodotta.

10.1 Introduzione

In questo capitolo illustreremo il testing della nostra selezione delle firme su due opportuni codici sorgente; in tale contesto evidenzieremo alcuni aspetti di rilievo e discuteremo i risultati ottenuti. Per questioni commerciali non possiamo specificare il vero nome dei codici sorgente; per tale ragione, nel seguito del capitolo, per indicare tali codici, useremo gli pseudonimi "Codice 1" e "Codice 2".

Le firme considerate, essendo diretta conseguenza delle linee guida di sviluppo in sicurezza, rivestono particolare importanza in qualsiasi processo di ispezione del codice e, per questo motivo, andranno sempre abilitate. L'applicazione di queste firme va, quindi, considerata con particolare attenzione.

La versione della suite software Fortify utilizzata è la 4.5.0.0337; con essa vengono fornite delle firme, la cui versione è la 4.5.0.0061, che hanno una validità di due mesi a partire dalla data di installazione.

Dopo aver importato il pacchetto di regole aggiornate siamo giunti alla versione delle regole 4.5.0.0063 che è quella da noi utilizzata.

10.2 Considerazioni relative all'attività di selezione delle firme

La fase di selezione delle firme evidenzia una particolare struttura la cui comprensione è importante ai fini di una corretta analisi di un codice sorgente.

La Figura 10.1 mostra sinteticamente il processo di code inspection ponendo attenzione al codice sorgente ed alla selezione delle firme di rilevamento.

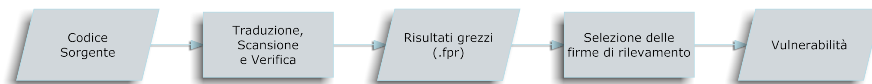


Figura 10.1. Selezione delle firme di rilevamento a partire dalle "best practices"

Si consideri un dato codice sorgente e lo si sottoponga a traduzione, scansione e verifica; i risultati grezzi saranno contenuti nel file `.fpr`.

Partendo da questi ultimi potremo far corrispondere alla selezione di una generica firma (relativa ad un dato analizzatore) una vulnerabilità ed un numero intero non negativo che indica l'occorrenza della vulnerabilità rilevata.

Sinteticamente:

$$r_i \xrightarrow{\quad g \quad} (v_i, n_i) \tag{10.1}$$

In Figura 10.2 viene fornita una rappresentazione della funzione g dall'insieme delle firme F nell'insieme (delle coppie) di vulnerabilità-occorrenze $\mathrm{V} \times \mathrm{N}$.

Se, ad esempio, selezionassimo le firme di rilevamento r_2, r_3 ed r_7 (r sta per *rule*) osserveremmo che le vulnerabilità con le relative occorrenze sarebbero (v_2, n_2), (v_3, n_3) e (v_7, n_7).

In tal caso potremmo scrivere:

$$f(r_2 \cup r_3 \cup r_7) = (v_2, n_2) \cup (v_3, n_3) \cup (v_7, n_7) \tag{10.2}$$

dove la nostra f rappresenterebbe un funzione dall'insieme delle parti dell'insieme delle firme selezionabili all'insieme delle parti dell'insieme delle vulnerabilità con le relative occorrenze.

In Figura 10.3 viene fornita una rappresentazione della funzione f appena descritta.

Se considerassimo, adesso, un elemento $a_i \in \{0, 1\}$, con $i = 1, 2, \ldots, m$ (dove m è il numero di firme totali per pacchetto), potremmo definire un'operazione esterna $\{0, 1\} \times \mathrm{F} \to \mathrm{F}$ del tipo:

$$a_i r_i = \begin{cases} \emptyset & \text{se } a_i = 0, \\ r_i & \text{se } a_i = 1. \end{cases} \tag{10.3}$$

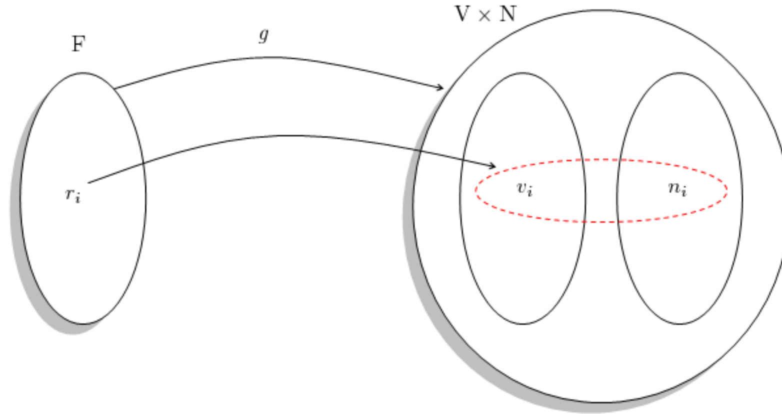


Figura 10.2. Funzione g

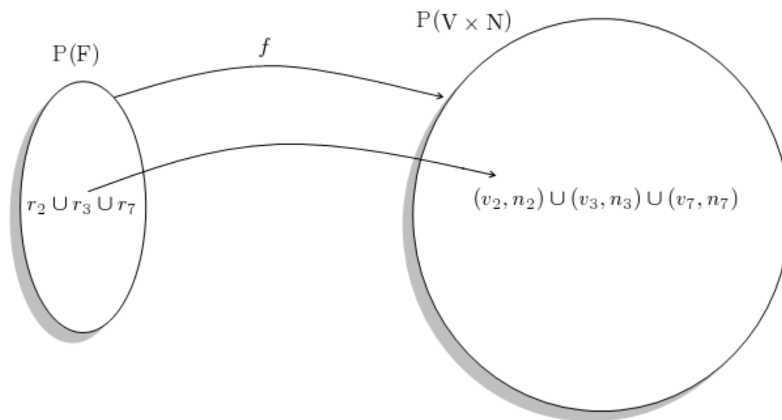


Figura 10.3. Funzione f

così che ogni elemento dell'insieme delle parti potrebbe essere rappresentato da:

$$a_1 r_1 \cup a_2 r_2 \cup \cdots \cup a_m r_m \tag{10.4}$$

Infatti, ad esempio, se volessimo rappresentare l'elemento costituito dall'insieme delle tre firme r_2, r_3 ed r_7, potremmo scrivere:

$$r_2 \cup r_3 \cup r_7 = 0r_1 \cup 1r_2 \cup 1r_3 \cup \cdots \cup 1r_7 \cup \cdots \cup 0r_m \cup \tag{10.5}$$

ovvero potremmo scrivere l'espressione 10.4 in cui solo i coefficienti a_2, a_3 ed a_7 hanno un valore pari ad 1.

Abbiamo, così, ottenuto:

$$f(a_1r_1 \cup a_2r_2 \cup \cdots \cup a_mr_m) = a_1(v_1, n_1) \cup a_2(v_2, n_2) \cup \cdots \cup a_m(v_m, n_m) \quad (10.6)$$

L'equazione 10.6 così ottenuta è molto importante in quanto rappresenta la logica di funzionamento "lineare" alla base della selezione delle firme di rilevamento.

L'introduzione del "bit" a_i permette di interpretare la selezione delle firme di rilevamento come la selezione di una stringa di bit. Tale interpretazione diventa chiara se si pensa che la selezione delle firme avviene tramite "checkbox" che, per definizione, possono assumere soltanto due valori. Una checkbox non spuntata indica una valore falso, associabile, in questo caso, alla mancata selezione della firma in questione. Una checkbox spuntata indica, invece, un valore vero, associabile, in questo caso, alla selezione della firma in questione.

Nel caso delle firme "Fortify Secure Coding Rules, Extended, C/C++" le firme sono 79 (non considerando i relativi package) avendo, così, $2^{79} \simeq 6 \times 10^{23}$ combinazioni di firme che possono essere selezionate.

Nel caso delle firme "Fortify Secure Coding Rules, Extended, Java" le firme sono 51 (non considerando i relativi package) avendo, così, $2^{51} \simeq 2,25 \times 10^{15}$ combinazioni di firme che possono essere selezionate.

10.3 Metodologia

Come già detto, il processo di Code Inspection si articola attraverso un percorso strutturato in sette fasi, distinte e successive, all'interno delle quali vengono effettuate le operazioni che porteranno man mano a restringere il campo di osservazione agli aspetti peculiari e rilevanti dell'applicazione sotto esame e a fornire gli elementi necessari ad interpretare le vulnerabilità rilevate. Tali elementi ne consentono una valutazione oggettiva e focalizzata alle principali funzionalità applicative presenti. Le fasi a cui si fa riferimento sono:

1. verifica formale della completezza del materiale e della documentazione;
2. caricamento del materiale sulla piattaforma di analisi;
3. individuazione delle aree di esposizione dell'applicazione;
4. selezione delle firme di rilevamento;
5. identificazione delle occorrenze e classificazione delle vulnerabilità tecnologiche rilevate;
6. contestualizzazione delle vulnerabilità;
7. organizzazione dei risultati.

Il "Codice 1" sarà oggetto di tutte le fasi, mentre il "Codice 2" sarà oggetto della sola fase relativa alla selezione delle firme.

10.4 Codice 1

Il software oggetto di studio indicato con lo pseudonimo di "Codice 1" è un'applicazione J2EE che si occupa della gestione di informazioni relative ad attività di vendita e post-vendita di utenti di telefonia mobile; risulta, quindi, evidente che le informazioni trattate sono da considerarsi particolarmente critiche dal punto di vista delle sicurezza.

10.4.1 Verifica formale della completezza del materiale e della documentazione

La prima attività del processo di ispezione del codice sorgente è l'acquisizione di informazioni relative all'applicativo software che siano complete e formalmente valide. A tal proposito ci si riferisce all'apposita check-list individuata dalla Tabella 4.1. Ricordiamo che i principali documenti forniti contestualmente all'applicativo sono il Piano di Sicurezza dell'Applicazione (PDS) ed il Piano di Rientro dell'Applicazione (PDR).

Le tipologie di informazioni a cui abbiamo volto particolare attenzione sono quelle necessarie alla compilazione del codice e quelle descrittive della sua architettura.

Tutte le informazioni relative all'applicativo, che non vengono qui descritte per motivi commerciali, sono state individuate e ritenute valide e sufficienti all'avvio delle successive attività di ispezione.

10.4.2 Caricamento del materiale sulla piattaforma di analisi

Dopo aver verificato la consistenza del materiale informativo ricevuto, abbiamo caricato il codice sorgente sulla piattaforma di analisi che, nel nostro caso, è costituita dal software Fortify. Il codice, quindi, è stato sottoposto a:

- traduzione;
- scansione;
- verifica.

Tutte le operazioni automatizzate sono state eseguite sulla base del fatto che il software sotto esame è un'applicazione J2EE.

Quindi, facendo riferimento a quanto scritto nella Sezione 5.4.1, le attività eseguite sono state:

- traduzione di file Java;
- traduzione di file JSP;
- traduzione di altri file a corredo (nel caso dell'applicativo in oggetto si tratta di file XML);
- elaborazione dei file di configurazione.

Sono state effettuate due analisi al fine di definire l'intera attività con una elevata accuratezza d'azione.

La prima parte della nostra esperienza è stata dedicata ad un'analisi non ottimizzata a causa di una non completa identificazione della struttura logica dell'applicativo analizzato, in particolar modo per quanto riguarda le classi Java utilizzate.

Successivamente, invece, è stata considerata un'analisi basata su un ulteriore approfondimento delle informazioni disponibili, in particolar modo dal punto di vista di una corretta identificazione della struttura logica del codice sorgente. L'identificazione della correttezza della seconda sessione di analisi è avvenuta anche in base all'individuazione delle differenze con la prima. Infatti, non è possibile, a priori, affermare che una sessione sia sicuramente corretta o efficace. Nel caso preso in esame, la conferma viene tanto dalla migliore definizione delle classi Java richieste quanto dall'individuazione di ulteriori problematiche rilevanti dal punto di vista della sicurezza.

Analisi non ottimizzata

Fase di Traduzione

Al fine di poter meglio identificare eventuali problematiche legate all'acquisizione di ciascun linguaggio utilizzato nell'applicativo, la fase di traduzione è stata effettuata in modo puntuale per i file Java, JSP, XML e `.properties`. Nei Listati 10.1, 10.2, 10.3 e 10.4 vengono riportati i comandi necessari per effettuare la traduzione del codice sorgente fornito alla piattaforma di analisi per i vari tipi di file.

```
sourceanalyzer -b Codice_1_s1 -cp
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\lib\*.jar"
 "C:\Programmi\Fortify Software\SCAS-EE4.5.0\sorgenti\Codice_1\**\*.java"
```

Listato 10.1. Traduzione non ottimizzata dei file Java del "Codice 1"

```
sourceanalyzer -b Codice_1_s1 -cp
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\lib\*.jar"
 "C:\Programmi\FortifySoftware\SCAS-EE4.5.0\sorgenti\Codice_1\**\*.jsp"
```

Listato 10.2. Traduzione non ottimizzata dei file JSP del "Codice 1"

```
sourceanalyzer -b Codice_1-s1 -cp
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\lib\*.jar"
 "C:\Programmi\Fortify Software\SCAS-EE4.5.0\sorgenti\Codice_1\**\*.xml"
```

Listato 10.3. Traduzione non ottimizzata dei file XML del "Codice 1"

```
sourceanalyzer -b Codice_1_s1 -cp
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\lib\*.jar"
 "C:\Programmi\Fortify Software\SCAS-EE4.5.0\sorgenti\Codice_1\**\*.properties"
```

Listato 10.4. Traduzione non ottimizzata dei file `.properties` del "Codice 1"

Fase di Scansione

I risultati dell'analisi sono contenuti nel file `codice_1_s1.fpr`. Si osservi l'utilizzo dell'opzione `-logfile` per la generazione del file di log che permette di visionare eventuali errori o punti di attenzione e di verificare il corretto completamento della fase di scansione. Di seguito si riporta il comando utilizzato per effettuare la scansione del codice sorgente fornito alla piattaforma di analisi.

```
sourceanalyzer -b Codice_1_s1 -scan -f codice_1_s1.fpr - logfile codice_1_s1.log
```

Analisi Ottimizzata

Fase di Traduzione

Così come per il caso precedente, la fase di traduzione è stata effettuata in modo puntuale per i file Java, JSP, XML e `.properties`. Nella linea di comando utilizzata vengono, però, indicate le classi Java utilizzate e identificate in seguito ad una ulteriore analisi e verifica della struttura logica dell'applicativo. Nei Listati 10.5, 10.6, 10.7 e 10.8 vengono riportati i comandi necessari per effettuare la traduzione del codice sorgente fornito alla piattaforma di analisi per i vari tipi di file.

```
sourceanalyzer -b Codice_1_s2 -cp
 "C:\apache-ant-1.7.0\lib\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libext\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\jars\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libCodice_1\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\**\*.java"
```

Listato 10.5. Traduzione ottimizzata dei file Java del "Codice 1"

```
sourceanalyzer -b Codice_1_s2 -cp
 "C:\apache-ant-1.7.0\lib\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libext\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\jars\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libCodice_1\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\**\*.jsp"
```

Listato 10.6. Traduzione ottimizzata dei file JSP del "Codice 1"

```
sourceanalyzer -b Codice_1_s2 -cp
 "C:\apache-ant-1.7.0\lib\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libext\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\jars\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libCodice_1\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\**\*.xml"
```

Listato 10.7. Traduzione ottimizzata dei file XML del "Codice 1"

```
sourceanalyzer -b Codice_1_s2 -cp
 "C:\apache-ant-1.7.0\lib\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libext\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\jars\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\libCodice_1\*.jar"
 "C:\Programmi\Fortify Software\SCASEE4.5.0\sorgenti\Codice_1\**\*.properties"
```

Listato 10.8. Traduzione ottimizzata dei file `.properties` del "Codice 1"

Fase di Scansione

I risultati dell'analisi si trovano nel file `codice_1_s2.fpr`. Anche in questo caso l'utilizzo dell'opzione `-logfile` è necessaria per la generazione del file di log sul quale visionare eventuali errori o punti di attenzione e per verificare il corretto completamento della fase di scansione. Di seguito si riporta la linea di comando necessaria per effettuare la scansione del codice sorgente fornito alla piattaforma di analisi.

```
sourceanalyzer -b Codice_1_s2 -scan -f codice_1_s2.fpr - logfile codice_1_s2.log
```

10.4.3 Attività di Analisi Preliminare

Terminata l'acquisizione del codice sorgente, si è avviata la fase di visualizzazione dei risultati attraverso Fortify Audit Workbench e, di conseguenza, quella della loro opportuna contestualizzazione sulla base di informazioni che possono avere carattere architetturale e funzionale o che possono essere strettamente legate al linguaggio utilizzato. L'attività di contestualizzazione è stata effettuata attraverso i seguenti passi:

- Associazione tra le aree di esposizione dell'applicazione, precedentemente identificate, e le famiglie di firme Fortify. Tale operazione ha rivestito un'estrema rilevanza al fine della corretta valutazione delle problematiche di sicurezza.
- Selezione delle firme di rilevamento.

Lo scopo primario dell'analisi preliminare consiste nel giungere ad una restrizione del campo di osservazione agli aspetti peculiari dell'applicazione sotto esame e nel fornire un'interpretazione delle problematiche rilevate che ne consenta una valutazione quanto più possibile oggettiva e focalizzata alle principali funzioni che questa utilizza. È stato, quindi, indispensabile procedere all'identificazione delle aree di esposizione dell'applicazione, dei suoi punti di contatto con il mondo esterno e, quindi, dei punti maggiormente esposti ad attacchi. L'obiettivo principale di questa attività è stato la predisposizione o, ancor meglio, il consolidamento dell'ambiente in previsione della successiva fase di analisi avanzata.

Identificazione delle aree di esposizione e di rischio

L'analisi delle interfacce e delle modalità di accesso alle funzionalità ha evidenziato l'impiego di un'interfaccia Web, scritta in JSP, per l'accesso alle informazioni da parte di operatori. Tale informazione viene considerata anche in sede di esame delle problematiche riscontrate dall'analisi del codice sorgente in quanto l'interazione "umana" potrebbe introdurre un ulteriore livello di rischio (ad esempio, con il tentativo di forzare il normale flusso operativo o con tentativi di accesso a risorse non autorizzate).

In base a tali considerazioni, nel corso delle successive fasi di analisi, si è prestata particolare attenzione alle problematiche classificate legate alla validazione delle informazioni in ingresso (Input Validation and Representation) ed alla possibilità di rivelare informazioni (Information Disclosure ed Information Leakage).

Relativamente al caso del "Codice 1", dunque, le principali informazioni di cui tener conto sono state le seguenti:

- l'impiego di un'interfaccia Web;
- l'utilizzo del linguaggio JAVA/JSP;
- l'elevata criticità delle informazioni trattate.

Individuazione dell'associazione tra le aree di esposizione e le famiglie Fortify

Prendendo in considerazione la Tabella 4.2, sulla base delle caratteristiche dell'applicazione (nel caso specifico un'applicazione J2EE), sono state selezionate ed evidenziate le macro-categorie, le categorie e le famiglie di vulnerabilità alle quali essa è esposta. A livello operativo, facendo riferimento alla Tabella 5.1, si è proceduto all'individuazione dell'associazione tra le aree di esposizione dell'applicazione identificate e le famiglie Fortify.

I domini identificati in base a tali presupposti sono dunque:

- Input Validation and Representation;
- Security Features.

Le categorie identificate, invece, sono:

- Input Validation;
- Cross Site Scripting;
- SQL Injection;
- Security Features;
- Poor Error Handling.

Importazione dell'Analisi Grezza

L'importazione dei risultati dell'analisi grezza ottimizzata è avvenuta attraverso l'apertura del file `.fpr` in Fortify Audit Workbench. Prima, però, è stato reso disponibile il codice sorgente sul sistema in cui ha avuto luogo la fase di analisi preliminare. Nel caso in cui il codice non fosse stato disponibile nella sua interezza, Fortify Audit Workbench avrebbe visualizzato esclusivamente le righe direttamente coinvolte nella problematica riscontrata.

Selezione delle firme di rilevamento

In generale, la selezione delle firme di rilevamento viene effettuata sulla base di un'opportuna considerazione dei seguenti elementi:

- linguaggio utilizzato;
- tipologia di risorse accedute;
- esposizione dell'applicazione/API;
- tipologia di attività effettuate.

Nel caso di nostro interesse, al fine di garantire l'individuazione di un range di problematiche quanto più ampio possibile, è stata utilizzata la selezione "ampia" (identificata da Fortify come "Broad"). Tale modalità operativa, in assenza di specifiche esigenze di analisi, derivanti, ad esempio, da conoscenze acquisite in seguito a Technical Security Audit, garantisce che non viene sottovalutata alcuna problematica; viene, così, demandato alle successive fasi di analisi il compito di scremare eventuali problematiche ritenute non rilevanti o classificate come falsi positivi.

In Figura 10.4 viene fornita una rappresentazione, tramite Fortify Audit Workbench, delle vulnerabilità identificate nel "Codice 1" con la selezione delle firme "Broad" di default per Fortify.

I risultati conseguiti, secondo la classificazione "tecnologica" utilizzata dalla piattaforma di analisi, sono i seguenti:

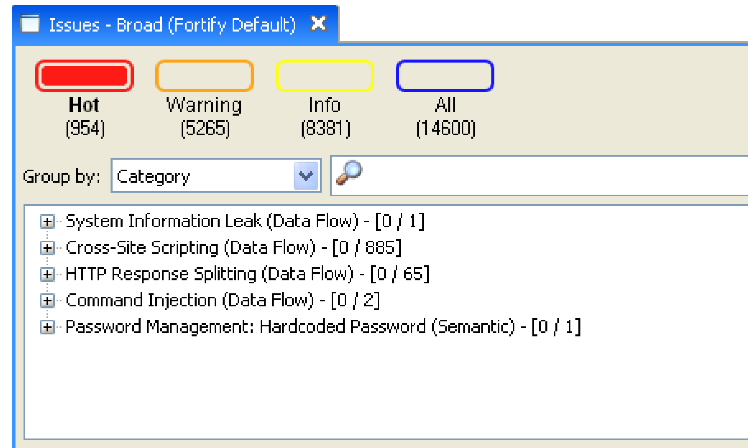


Figura 10.4. Vulnerabilità identificate nel "Codice 1" con selezione delle firme "Broad" di default per Fortify

- "Hot": 954 occorrenze di vulnerabilità;
- "Warning": 5265 occorrenze di vulnerabilità;
- "Info": 8381 occorrenze di vulnerabilità;

per un totale di 14600 occorrenze di vulnerabilità.

In termini di specifiche categorie Fortify, le problematiche individuate e considerate "Hot" e, quindi, di maggiore rilevanza sono:

- *Cross-Site Scripting*: 885 occorrenze;
- *HTTP Response Splitting*: 65 occorrenze;
- *Command Injection*: 2 occorrenze;
- *Password Management*: 1 occorrenza;
- *System Information Leak*: 1 occorrenza.

In Figura 10.5 viene fornita una rappresentazione delle vulnerabilità identificate nel "Codice 1" riscontrate attraverso la selezione delle firme "Medium" di default per Fortify. In Figura 10.6 viene, invece, fornita una rappresentazione delle vulnerabilità identificate nel "Codice 1" riscontrate attraverso la selezione delle firme "Targeted" di default per Fortify.

Esportazione dei filtri e configurazione

Immediatamente dopo aver controllato il risultato derivante dall'attuale selezione di firme, è stato creato ed esportato il relativo filtro; questo consiste in un file di testo contenente le opportune impostazioni. Tale file potrà essere successivamente

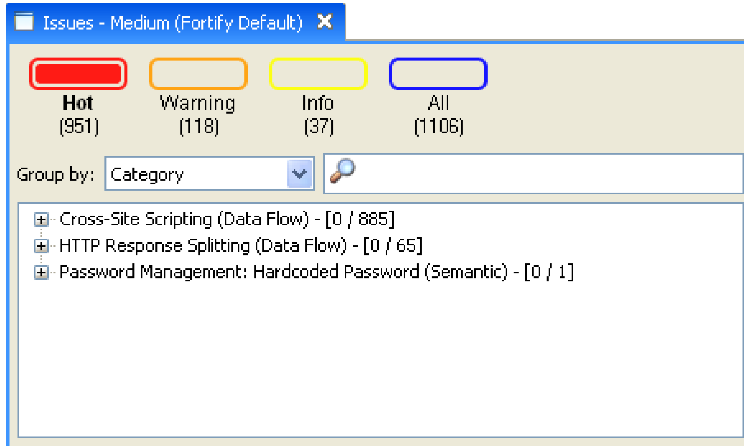


Figura 10.5. Vulnerabilità identificate nel "Codice 1" con selezione delle firme "Medium" di default per Fortify

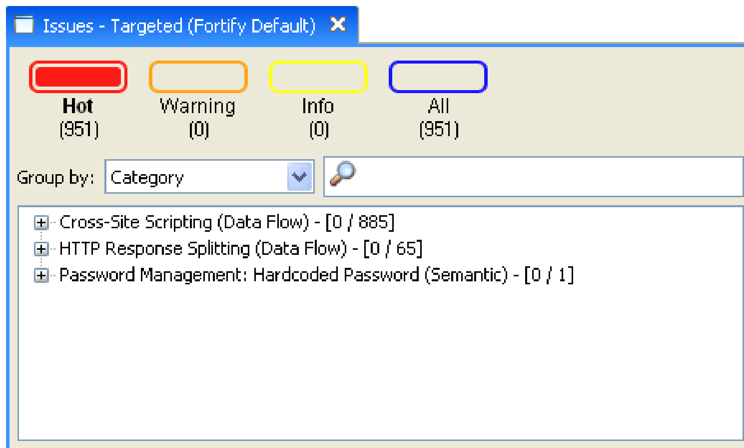


Figura 10.6. Vulnerabilità identificate nel "Codice 1" con selezione delle firme "Targeted" di default per Fortify

utilizzato sia in fase di ulteriore analisi sia in fase di scansione con Fortify SCA specificando l'opzione `-filter` seguita dal nome del file di filtro, come di seguito riportato:

```
sourceanalyzer -b <build-id> -scan -f result.fpr -filter filtro.txt
```

Salvataggio dell'Analisi Preliminare

Infine il progetto è stato salvato utilizzando la voce "Save Project As", presente all'interno del sottomenù "File". Utilizzando un nome diverso per ciascuna fase di analisi è possibile ricostruire con precisione le diverse fasi di interpretazione e configurazione.

10.4.4 Attività di Analisi Avanzata

In questa fase abbiamo identificato e valutato la criticità dei risultati ottenuti dall'acquisizione del codice sorgente e dall'Analisi Preliminare dello specifico applicativo. Sono stati validati tutti i riscontri emersi; questi sono stati successivamente classificati dal punto di vista tecnologico e del contesto.

L'attenzione si è focalizzata sulla problematica di "Cross-Site Scripting"; le ragioni sono piuttosto chiare in quanto si tratta di una problematica classificata dalla piattaforma di analisi come "Hot" e, dato indubbiamente importante, presente in ben 885 punti di ingresso.

Il Cross-Site Scripting (o XSS) è una vulnerabilità che affligge siti Web con scarso o non sufficiente controllo di variabili derivate da input dell'utente (spesso variabili GET). L'XSS permette di inserire codice a livello browser (spesso codice javascript pericoloso) al fine di modificare il codice sorgente della pagina Web visitata. In questo modo un cracker può tentare di recuperare dati sensibili, quali i cookie.

Dobbiamo evidenziare che questo, comunque, non deve allarmare né essere considerato ancora come risultato finale. Deve, comunque, essere acquisito come elemento ad elevata criticità. A tale scopo, una considerazione fondamentale è emersa tra le varie osservazioni effettuate: la problematica è stata individuata nel contesto di un applicativo che sfrutta e fa riferimento a dati altamente sensibili (dati di carattere anagrafico e commerciale) e che presenta tra le sue interfacce principali quella Web (tipologia di interfaccia, per sua natura, altamente esposta a diverse problematiche di sicurezza).

Si è, dunque, approfondito lo studio della problematica per accertarne l'effettiva rilevanza all'interno del codice sorgente, verificando che fossero applicabili le condizioni indicate dallo strumento e dalla descrizione della firma associata, seguendo, dove necessario, il flusso dell'algoritmo, i salti condizionali e tutte le chiamate ad altre funzioni coinvolte nella sezione incriminata. Solo successivamente, in base a tale analisi, si è proceduto all'assegnazione dello stato di "vulnerabilità confermata".

È stata, inoltre, valutata l'importanza della funzione all'interno della quale questa è stata trovata rispetto alle funzionalità e al comportamento atteso dell'applicazione. In questo modo è possibile discriminare tra le vulnerabilità che non possono in nessun modo portare danni al funzionamento del sistema, ad esempio perché non esposte o perché agiscono verso sistemi passivi mono-direzionali,

quali stampanti o dischi ottici riscrivibili, e quelle che invece espongono l'applicazione e i dati che essa tratta a danni gravi, quali interruzioni, alterazioni del comportamento o dei dati.

In ultimo, un passo principale dell'attività di Analisi Avanzata, come già indicato, risulta essere quello della valutazione della problematica rispetto al contesto generale dell'applicativo. Per classificare la rilevanza delle vulnerabilità riscontrata è stata utilizzata la classificazione riportata in Tabella 4.3, mentre per la sua categorizzazione rimane valido l'albero delle vulnerabilità che corrisponde alle aree di esposizione dell'applicazione riportate in Tabella 4.2.

Invio dei risultati a Fortify Security Manager

Ai fini della gestione e dell'estrazione della reportistica e ai fini della storicizzazione dell'attività, l'ultima fase del processo si occupa dell'invio dei risultati ottenuti a Fortify Security Manager. È opportuno sottolineare come sia stato necessario utilizzare i medesimi accorgimenti e le medesime precauzioni di sicurezza e riservatezza utilizzati per la gestione del codice sorgente per tutte le informazioni raccolte ed utilizzate in sede di Code Inspection.

La Figura 10.7 mostra la visualizzazione del progetto tramite Fortify Security Manager.

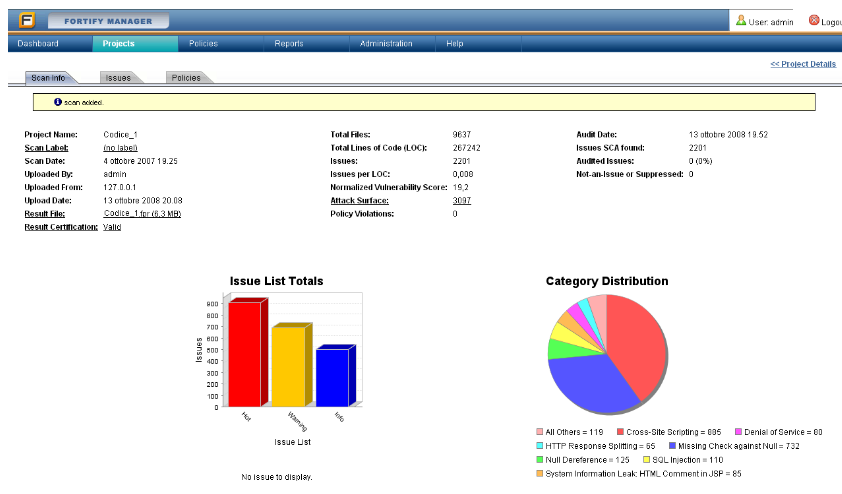


Figura 10.7. Visualizzazione del progetto tramite Fortify Security Manager

10.4.5 Audit con selezione delle firme relative alle "best practices" per il "Codice 1"

In questa sottosezione verrà effettuata l'analisi dei risultati dell'applicazione delle firme relative alle "best practices" al "Codice 1". Da notare che questa sessione è stata effettuata abilitando tutte le firme relative al pacchetto "Fortify Secure Coding Rules, Extended, JSP" come mostrato in Figura 10.8.

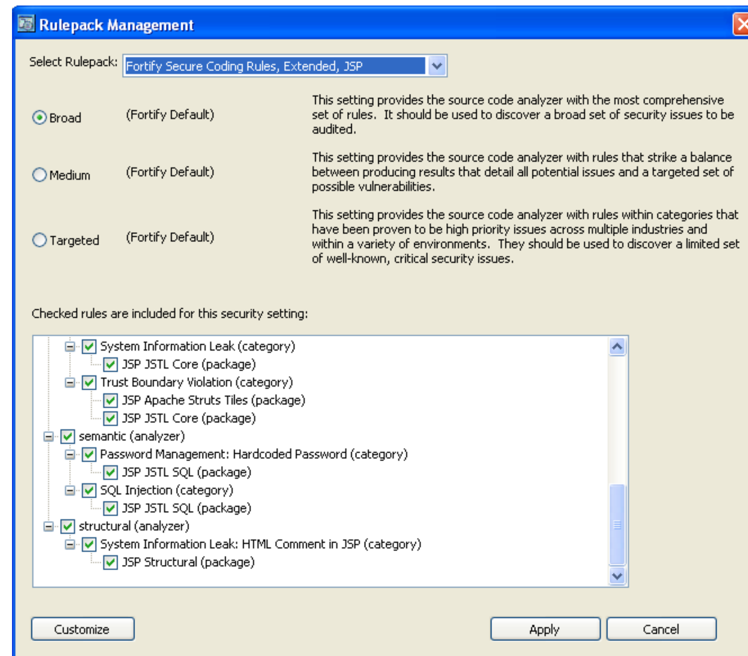


Figura 10.8. Abilitazione di tutte le firme del pacchetto "Fortify Secure Coding Rules, Extended, JSP"

In Figura 10.9, invece, viene fornita una rappresentazione, tramite Fortify Audit Workbench, delle vulnerabilità identificate nel "Codice 1", considerate "Hot" e riscontrate attraverso la selezione delle firme relativa alle "best practices".

I risultati conseguiti relativamente a tale selezione sono:

- "Hot": 950 occorrenze di vulnerabilità;
- "Warning": 725 occorrenze di vulnerabilità;
- "Info": 526 occorrenze di vulnerabilità.

per un totale di 2201 occorrenze di vulnerabilità.

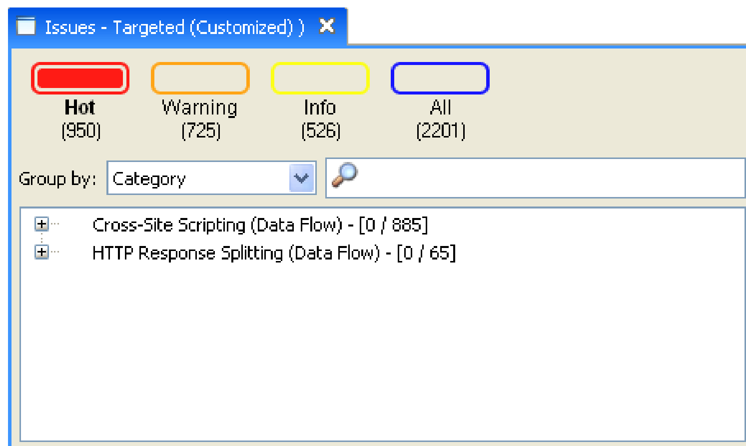


Figura 10.9. Vulnerabilità identificate nel "Codice 1" considerate "Hot" con la selezione delle firme relativa alle "best practices"

In termini di specifiche categorie Fortify, le problematiche individuate e considerate "Hot" e, quindi, di maggiore rilevanza sono:

- *Cross-Site Scripting*: 885 occorrenze;
- *HTTP Response Splitting*: 65 occorrenze.

Nelle Figure 10.10 e 10.11 vengono rappresentate, tramite Fortify Audit Workbench, rispettivamente le vulnerabilità identificate considerate "Warning" e quelle identificate considerate "Info" nel "Codice 1", riscontrate attraverso la selezione delle firme relativa alle "best practices".

Soffermeremo la nostra attenzione sulle vulnerabilità considerate "Hot" e confronteremo i risultati ottenuti con la selezione "Broad" di default per Fortify, che contempla la selezione di tutte le firme di tutti i pacchetti, con quelli ottenuti dalla selezione delle firme dei pacchetti Java considerata nel Capitolo 9.

Risulta subito evidente che il numero di problematiche individuate con quest'ultima selezione di firme è inferiore. Ad ogni modo, quasi la totalità delle vulnerabilità considerate ad alto impatto sulla sicurezza dell'applicativo, ovvero Cross-Site Scripting e HTTP Response Splitting, sono correttamente identificate. Osserviamo, però, che non vengono individuate quattro vulnerabilità che, invece, sono presenti con la selezione "Broad". In particolare, non sono presenti:

- due occorrenze di Command Injection;
- un'occorrenza di Password Management;
- un'occorrenza di System Information Leak.

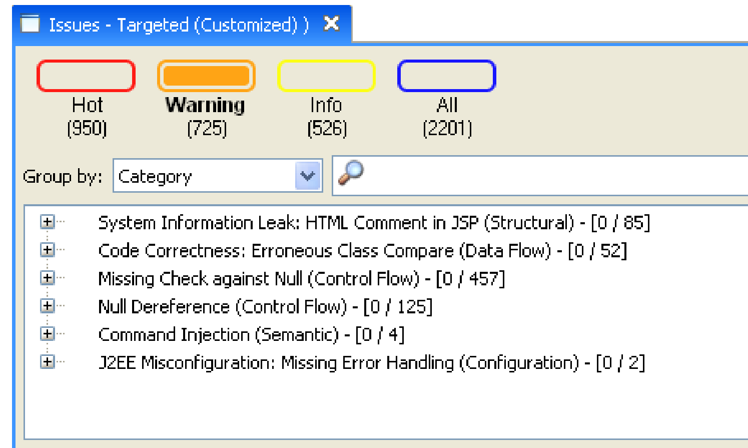


Figura 10.10. Vulnerabilità identificate nel "Codice 1" considerate "Warning" con la selezione delle firme relativa alle "best practices"

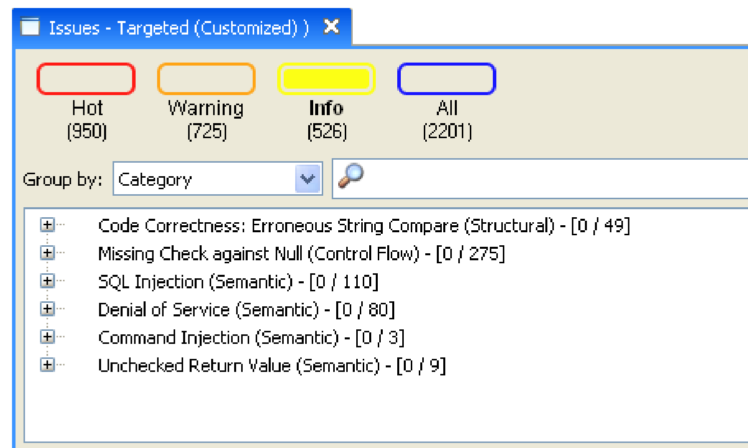


Figura 10.11. Vulnerabilità identificate nel "Codice 1" considerate "Info" con la selezione delle firme relativa alle "best practices"

Salta, così, subito all'occhio che le due occorrenze di Command Injection non vengono rilevate con la nostra selezione di firme nonostante siano abilitate le firme relative a questa vulnerabilità (di tutti gli analizzatori). Dopo alcune prove sperimentali è emerso che ciò è dovuto alla disabilitazione della firma "Java Properties" che è tra quelle etichettate con il termine "inputsource". La Figura 10.12 evidenzia con un segno di spunta di colore rosso la selezione della firma "Java Properties" che dovrebbe essere abilitata affinché vengano rilevate le occorrenze

di Command Injection di tipo "Hot" nel "Codice 1".

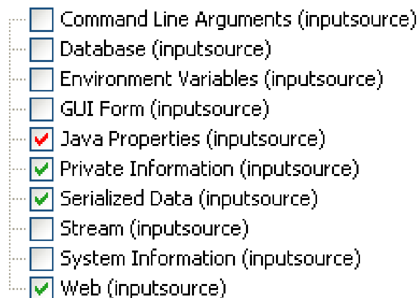


Figura 10.12. Selezione della firma "Java Properties"

La Tabella 10.1 riassume numericamente le occorrenze delle vulnerabilità riscontrate con le differenti selezioni per il "Codice 1"; basandosi su tali numeri è stato prodotto un grafico comparativo mostrato in Figura 10.13.

Livello	*Broad*	*Medium*	*Targeted*	*Best Practices*
Hot	954	951	951	950
Warning	5265	262	0	725
Info	8381	7874	0	526
Totali	14600	9087	951	2201

Tabella 10.1. Vulnerabilità riscontrate con varie selezioni per il "Codice 1"

Da questo si evince che la selezione relativa alle "best practices" mostra una pressoché identica rilevazione delle vulnerabilità, considerate da Fortify "Hot", rispetto alle selezioni "Broad", "Medium" e "Targeted" di default per Fortify. La selezione relativa alle "best practices" mostra una maggiore rilevazione delle vulnerabilità considerate da Fortify "Warnings" rispetto alla selezione "Medium" di default per Fortify, ma una minore rilevazione rispetto alla selezione "Broad" di default per Fortify. La selezione relativa alle "best practices" mostra una minore rilevazione delle vulnerabilità considerate da Fortify "Info" rispetto alle selezioni "Broad" e "Medium" di default per Fortify.

Infine, osserviamo che i risultati conseguiti secondo la selezione delle firme relative alle "best practices" ottenuti disabilitando tutte le firme del pacchetto "Fortify Secure Coding Rules, Extended, JSP" sono:

- "Hot": 950 occorrenze di vulnerabilità;

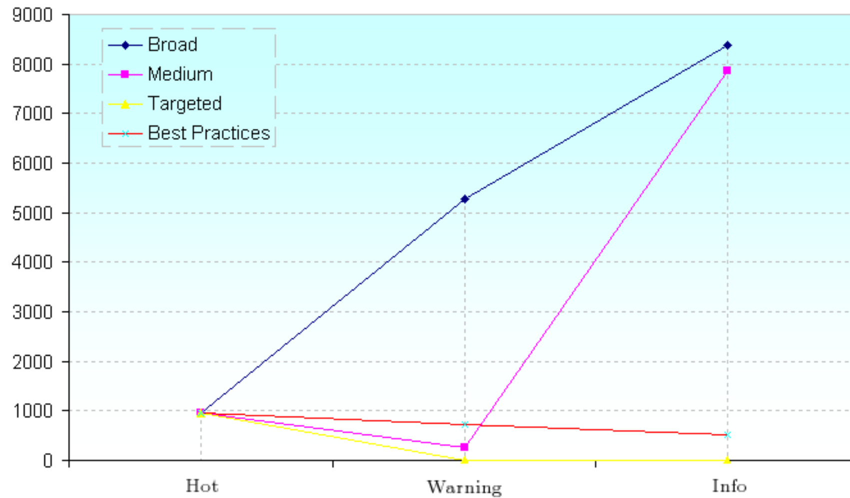


Figura 10.13. Grafico comparativo delle selezioni per il "Codice 1"

- "Warning": 640 occorrenze di vulnerabilità;
- "Info": 526 occorrenze di vulnerabilità;

per un totale di 2116 occorrenze di vulnerabilità.

Poiché il "Codice 1" è costituito anche da un'interfaccia scritta in JSP era prevedibile che, disabilitando tutte le firme relative a tale linguaggio, si sarebbe riscontrata una minore rilevazione di vulnerabilità rispetto al caso in cui le firme in questione fossero abilitate.

10.5 Codice 2

Anche per il "Codice 2", al fine di garantire l'individuazione di un range di problematiche quanto più ampio possibile, è stata utilizzata la selezione "ampia", ovvero la selezione in cui risultano abilitate tutte le firme di tutti i pacchetti disponibili sulla nostra piattaforma di analisi.

Tale modalità operativa, in assenza di specifiche esigenze di analisi, derivanti, ad esempio, da conoscenze acquisite in seguito a Technical Security Audit, garantisce di non sottovalutare alcuna problematica; viene, così, demandato alle successive fasi di analisi il compito di scremare eventuali problematiche ritenute non rilevanti o classificate come falsi positivi.

In Figura 10.14 viene fornita una rappresentazione delle vulnerabilità identificate nel "Codice 2" con selezione delle firme "Broad" di default per Fortify.

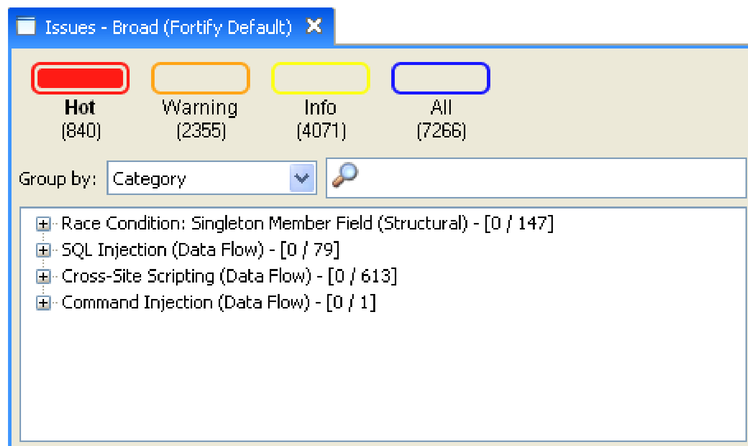


Figura 10.14. Vulnerabilità identificate nel "Codice 2" con selezione delle firme "Broad" di default per Fortify

I risultati conseguiti secondo la classificazione "tecnologica" utilizzata dalla piattaforma di analisi sono i seguenti:

- "Hot": 840 occorrenze di vulnerabilità;
- "Warning": 2355 occorrenze di vulnerabilità;
- "Info": 4071 occorrenze di vulnerabilità;

per un totale di 7266 occorrenze di vulnerabilità.

In termini di specifiche categorie Fortify, le problematiche individuate e considerate "Hot" e, quindi, di maggiore rilevanza sono:

- *Race Condition: Singleton Member Field (Structural)*: 147 occorrenze;
- *SQL Injection (Data Flow)*: 79 occorrenze;
- *Cross-Site Scripting (Data Flow)*: 613 occorrenze;
- *Command Injection (Data Flow)*: 1 occorrenza.

In Figura 10.15 viene fornita una rappresentazione delle vulnerabilità identificate nel "Codice 2" riscontrate attraverso la selezione delle firme "Medium" di default per Fortify. In Figura 10.16 viene, invece, fornita una rappresentazione delle vulnerabilità identificate nel "Codice 2" riscontrate attraverso la selezione delle firme "Targeted" di default per Fortify.

10.5.1 Audit con selezione delle firme relative alle "best practices" per il "Codice 2"

In questa sottosezione viene effettuata l'analisi dei risultati dell'applicazione al "Codice 2" delle firme relative alle "best practices". Da notare che questa sessione

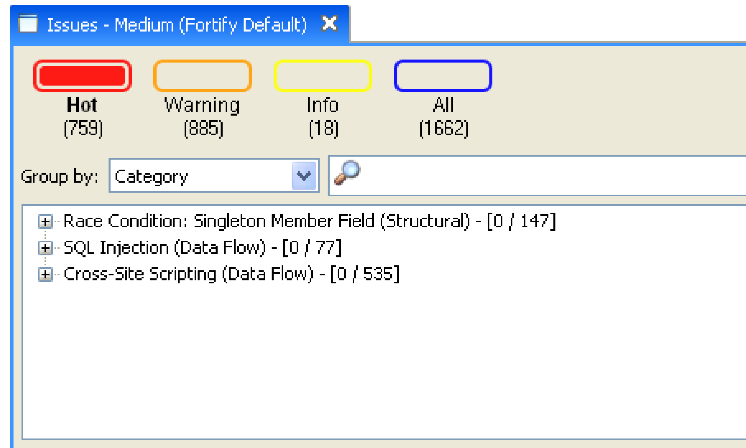


Figura 10.15. Vulnerabilità identificate nel "Codice 2" con selezione delle firme "Medium" di default per Fortify

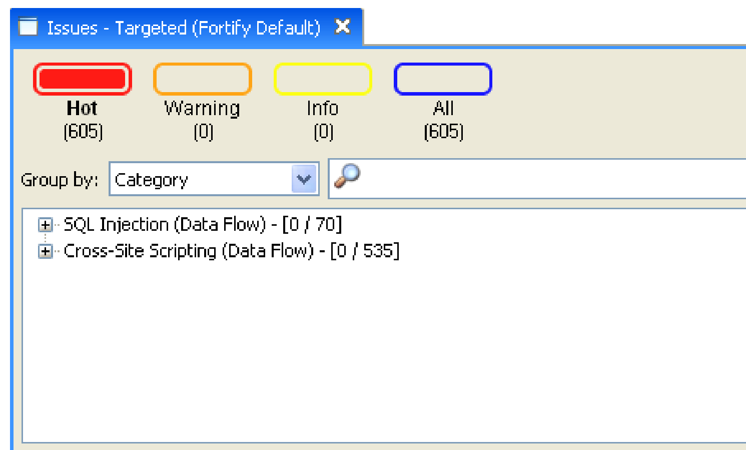


Figura 10.16. Vulnerabilità identificate nel "Codice 2" con selezione delle firme "Targeted" di default per Fortify

è stata effettuata abilitando tutte le firme relative al pacchetto "Fortify Secure Coding Rules, Extended, JSP", come mostrato in Figura 10.8.

In Figura 10.17 viene fornita una rappresentazione, tramite Fortify Audit Workbench, delle vulnerabilità identificate nel "Codice 2", considerate "Hot" e riscontrate attraverso la selezione delle firme relativa alle "best practices".

I risultati conseguiti, relativamente a tale selezione, sono i seguenti:

- "Hot": 605 occorrenze di vulnerabilità;

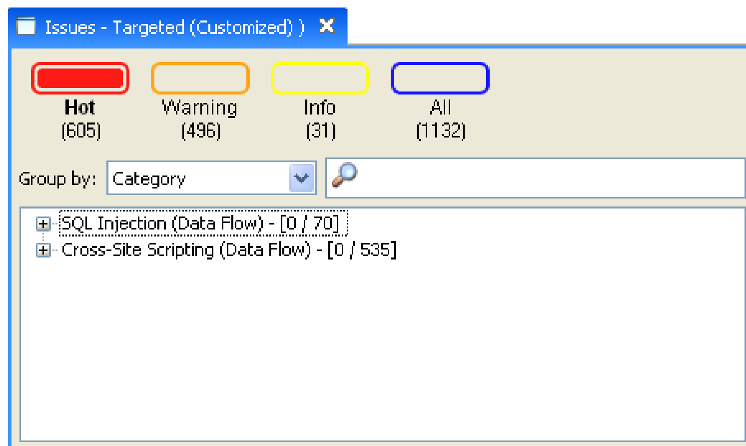


Figura 10.17. Vulnerabilità identificate nel "Codice 2" considerate "Hot" con la selezione delle firme relativa alle "best practices"

- "Warning": 496 occorrenze di vulnerabilità;
- "Info": 31 occorrenze di vulnerabilità;

per un totale di 1132 occorrenze di vulnerabilità riscontrate.

In termini di specifiche categorie Fortify, le problematiche individuate e considerate "Hot" e, quindi, di maggiore rilevanza sono:

- *SQL Injection (Data Flow)*: 70 occorrenze;
- *Cross-Site Scripting (Data Flow)*: 535 occorrenze.

In Figura 10.18 viene fornita una rappresentazione, tramite Fortify Audit Workbench, delle vulnerabilità identificate considerate "Warning" nel "Codice 2" con la selezione delle firme relativa alle "best practices". In Figura 10.19 viene fornita una rappresentazione, tramite Fortify Audit Workbench, delle vulnerabilità identificate considerate "Info" nel "Codice 2" con la selezione delle firme relativa alle "best practices".

Anche in questo caso soffermeremo la nostra attenzione sulle vulnerabilità considerate "Hot" e confronteremo i risultati ottenuti con la selezione "Broad" di default per Fortify, che contempla la selezione di tutte le firme di tutti i pacchetti, con quelli ottenuti con la selezione delle firme dei pacchetti Java considerata nel Capitolo 9.

Si nota subito che il numero di problematiche individuate con quest'ultima selezione di firme è inferiore. Comunque, la maggioranza delle vulnerabilità considerate ad alto impatto sulla sicurezza dell'applicativo, ovvero SQL Injection e Cross-Site Scripting, sono correttamente identificate. Osserviamo, però, che non

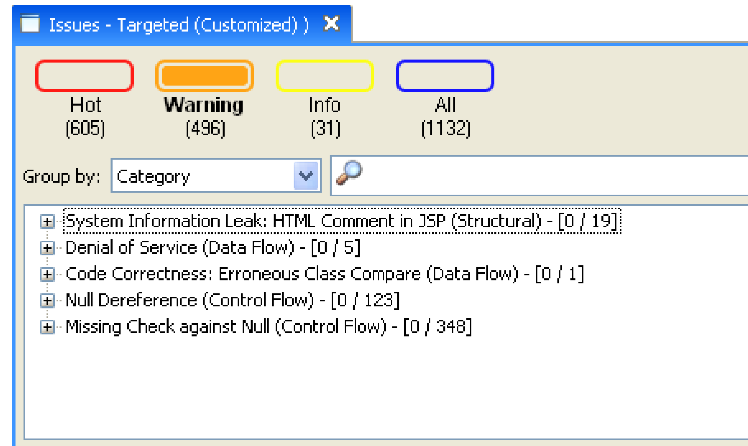


Figura 10.18. Vulnerabilità identificate nel "Codice 2" considerate "Warning" con la selezione delle firme relativa alle "best practices"

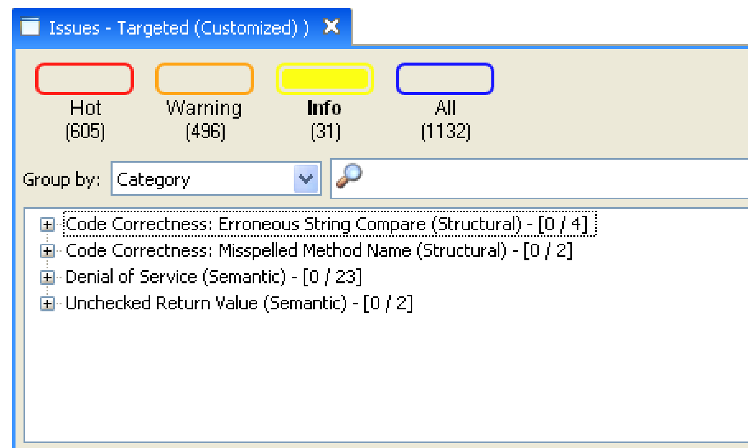


Figura 10.19. Vulnerabilità identificate nel "Codice 2" considerate "Info" con la selezione delle firme relativa alle "best practices"

vengono individuate 235 vulnerabilità che, invece, sono presenti con la selezione "Broad". In particolare, non sono presenti:

- 147 occorrenze di Race Condition: Singleton Member Field (Structural);
- 9 occorrenze di SQL Injection (Data Flow);
- 78 occorrenze di Cross-Site Scripting (Data Flow);
- un'occorrenza di Command Injection.

In questo caso l'occorrenza di Command Injection, come anche le altre non identificate, non dipende dall'abilitazione della firma "Java Properties"; infatti, la sua abilitazione non ha prodotto alcun cambiamento nella rilevazione delle occorrenze.

La Tabella 10.2 riassume le vulnerabilità riscontrate con varie selezioni per il "Codice 2"; basandosi su tali numeri è stato prodotto un grafico comparativo mostrato in Figura 10.20.

Livello	*Broad*	*Medium*	*Targeted*	*Best Practices*
Hot	840	759	605	605
Warning	2355	885	0	496
Info	4071	18	0	31
Totali	7266	1662	605	1132

Tabella 10.2. Vulnerabilità riscontrate con varie selezioni per il "Codice 2"

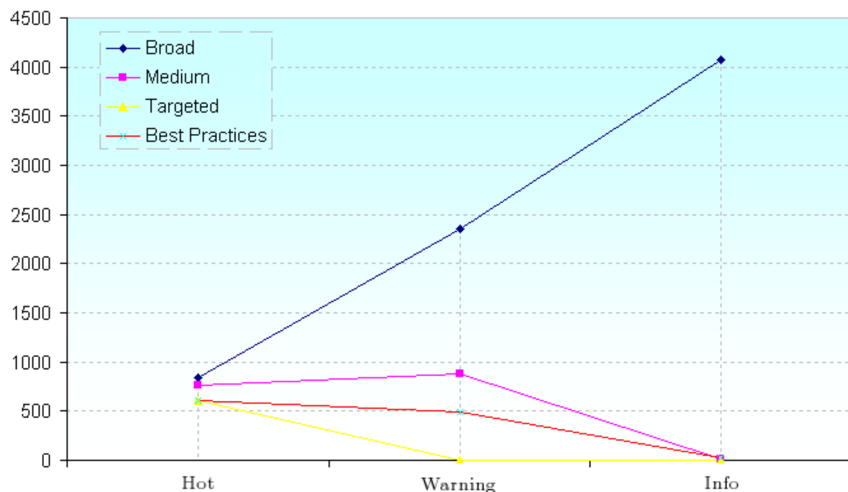


Figura 10.20. Grafico comparativo delle selezioni per il "Codice 2"

Dal grafico si evince che, nel caso del "Codice 2", la selezione relativa alle "best practices" mostra una minore rilevazione delle vulnerabilità considerate da Fortify "Hot", "Warnings" e "Info" rispetto alle selezioni "Broad" e "Medium" di default per Fortify.

Da quanto emerso da entrambi i grafici comparativi si può supporre, anche in base alle considerazioni matematiche espresse nella Sezione 10.2, che le firme comprese nel pacchetto "Targeted" di default per Fortify hanno come immagini degli elementi le cui vulnerabilità sono classificate da Fortify di gravità "Hot".

Infine, facciamo osservare che i risultati conseguiti secondo la selezione delle firme relative alle "best practices" ottenuti disabilitando tutte le firme del pacchetto "Fortify Secure Coding Rules, Extended, JSP" sono:

- "Hot": 605 occorrenze di vulnerabilità;
- "Warning": 477 occorrenze di vulnerabilità;
- "Info": 31 occorrenze di vulnerabilità;

per un totale di 1113 occorrenze di vulnerabilità.

Poiché anche il "Codice 2" è costituito da codice JSP era prevedibile che, disabilitando tutte le firme relative a tale linguaggio, si sarebbe riscontrata una minore rilevazione di vulnerabilità rispetto al caso in cui le firme in questione fossero abilitate. In questo caso la riduzione di identificazioni si presenta solo sul livello di gravità "Warning" con 19 occorrenze in meno.

Parte IV

Conclusioni

Quest'ultima parte chiude la presente tesi. Nel Capitolo 11 verranno tratte le opportune conclusioni e nel Capitolo 12 verranno menzionati alcuni possibili scenari futuri.

11

Conclusioni

Negli ultimi anni la crescita dei calcolatori e delle tecnologie dell'informazione ha avuto un carattere esplosivo. Anche questa nuova realtà è stata accompagnata da una forte esigenza di sicurezza. Infatti, i sistemi informatici e le reti di calcolatori sono divenuti obiettivo di attacco, di compromissione dei dati o di cattura delle informazioni per i più disparati motivi, da quelli di carattere strettamente economico a quelli di semplice carattere distruttivo.

Le nuove frontiere della sicurezza delle informazioni sono sempre più avanzate dal punto di vista tecnologico. Recentemente, agli strumenti quali crittografia, firewall e reti private virtuali si sono affiancati i software di analisi del codice sorgente (SCA, *Source Code Analyzers*).

Le funzionalità di un sistema di Code Inspection ruotano attorno alla selezione delle firme di rilevamento delle vulnerabilità ovvero all'abilitazione di un insieme di regole opportunamente valutate.

Il lavoro relativo alla presente tesi si colloca in questo particolare contesto. Grazie alla collaborazione dell'azienda ACSI Informatica è stato possibile rapportarsi con la sicurezza informatica dei sistemi in un'azienda leader nazionale nel campo delle telecomunicazioni.

In modo strettamente relazionato alle caratteristiche dell'ambiente di riferimento, l'attività ha avuto come obiettivi lo studio e la selezione di opportune firme di rilevamento.

Tra i vari prodotti software presenti sul mercato atti ai nostri scopi, lo strumento di analisi che meglio risponde alle attuali esigenze di questo segmento di mercato è costituito dalla suite software Fortify; per questo motivo tale suite è stata utilizzata per perseguire gli scopi del nostro lavoro e rappresenta la soluzione di più largo impiego tra le più importanti società del globo.

La selezione delle firme di rilevamento delle vulnerabilità è avvenuta in due fasi. La prima si è occupata della selezione delle firme di rilevamento che rispecchiano le "best practices" dei linguaggi di programmazione C/C++; la seconda, invece,

si è occupata della selezione delle firme di rilevamento che rispecchiano le “best practices” del linguaggio di programmazione Java.

Il lavoro si è concluso con la validazione delle firme selezionate tramite l’abilitazione delle stesse nelle attività di auditing su due codici sorgente Java/JSP.

12

Uno sguardo al futuro

I progressi tecnologici del software sono paralleli a quelli dell'hardware; spesso è difficile capire se tali progressi siano dettati da esigenze di costume o se queste ultime siano indotte dai primi. Di sicuro anche il contesto della sicurezza informatica è stato e sarà caratterizzato da continue evoluzioni. La frequente scoperta di nuove vulnerabilità presenti nei sistemi provoca, inevitabilmente, la necessità di un continuo aggiornamento delle soluzioni esistenti.

In termini di prossimi sviluppi, il primo passo consiste certamente nella definizione e nell'applicazione puntuale di firme di rilevamento opportune e, di conseguenza, in una loro valutazione in termini di gestione e di risultati. Solo attraverso un completo e quotidiano confronto con un contesto di grandi dimensioni, quale è stato quello di nostro riferimento, è possibile ottenere risultati specifici in termini di validità ed efficienza.

In un futuro più lontano, ma non tanto, l'adozione di protocolli come l'IPv6, l'introduzione di piattaforme software come l'attuale Android e la crescente esigenza di risparmio energetico cambieranno gli scenari a cui oggi siamo abituati; con ogni probabilità tutto ciò farà si che anche gli elettrodomestici saranno gestiti da veri e propri sistemi operativi e saranno connessi in rete.

Fra qualche anno non sarà utopia preoccuparsi del fatto che il nostro frigorifero possa essere infettato da un virus informatico compromettendone il suo normale, anche se sofisticato, funzionamento.

(Se il progresso aumenta le nostre preoccupazioni, e quindi anche il nostro stress, è vero "progresso"? Questa è un'altra storia :))

In quest'ottica, le attività di analisi del codice sorgente non solo saranno adottate dalle grandi società dell'informazione per la protezione dei propri sistemi informatici, ma estenderanno il loro campo d'azione al settore mobile, domotico e dei trasporti.

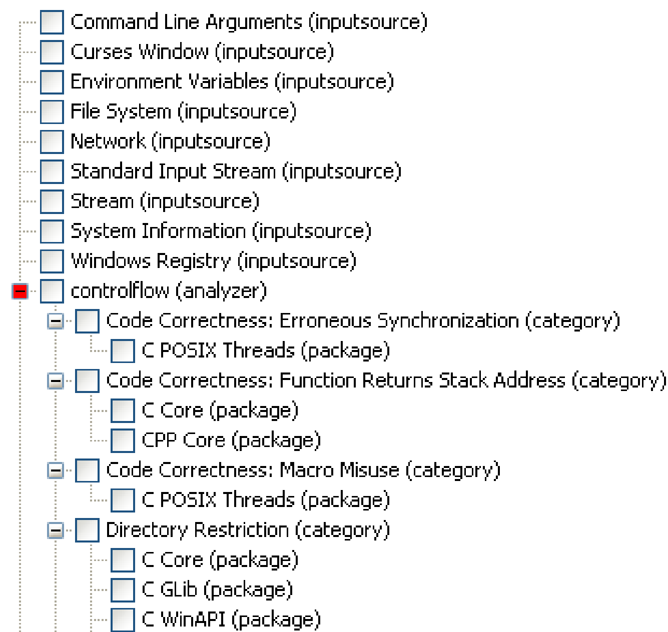
Riferimenti bibliografici

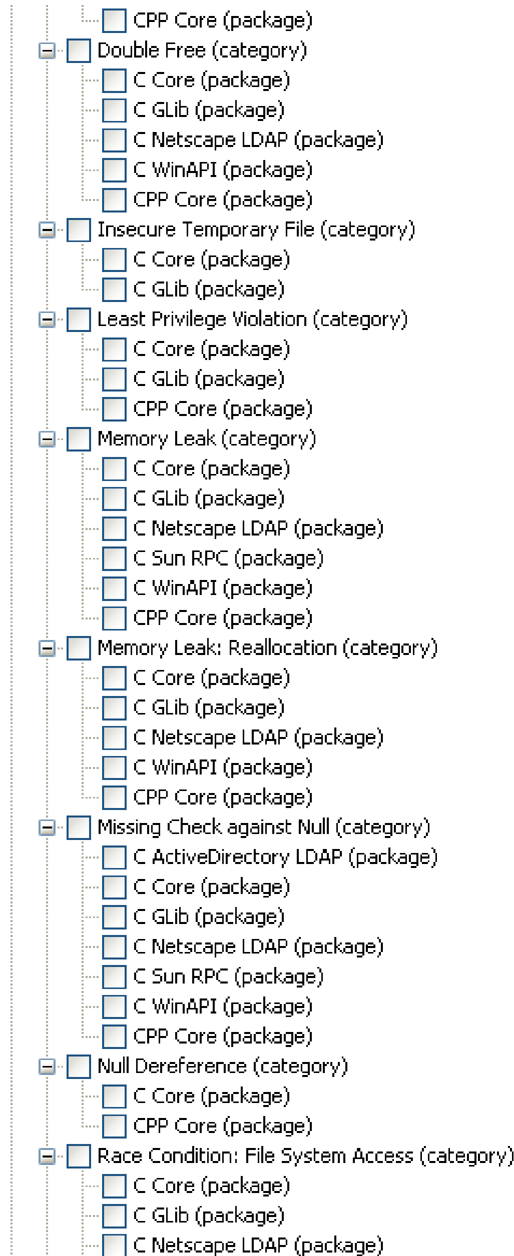
1. S. Bosworth and M.E. Kabay. *Computer Security Handbook*. John Wiley & Sons, fourth edition.
2. M.P. Cangemi and T. Singleton. *Managing the audit function: A corporate audit department procedures guide*. John Wiley & Sons, third edition.
3. CISSP. *Student Guide*. Thomson NETg, first edition.
4. M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15:182–211, 1976.
5. M. Gallagher. *Business Continuity Management*. Prentice Hall, 2005.
6. M. G. Graff and Kenneth R. van Wyk. *Secure Coding: Principles & Practices*. O'Reilly, 2003.
7. M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, second edition.
8. SANS Institute. Website. http://www.sans.org.
9. E. Maiwald and W. Sieglen. *Security planning & disaster recovery*. McGraw-Hill/Osborne, 2006.
10. M. Messier and J. Viega. *Secure Programming Cookbook for C and C++*. O'Reilly, 2003.
11. Y.F. Musaji. *Auditing and Security*. John Wiley & Sons, 2003.
12. M. Stamp. *Information security principles and practice*. John Wiley & Sons, 2006.
13. H.F. Tipton and M. Krause. *Information security management handbook*. CRC Press LLC, fifth edition.
14. M. Wallace and L. Webber. *The Disaster Recovery Handbook: A step-by-step plan to ensure business continuity and protect vital operations, facilities, and assets*. Amacom, 2004.

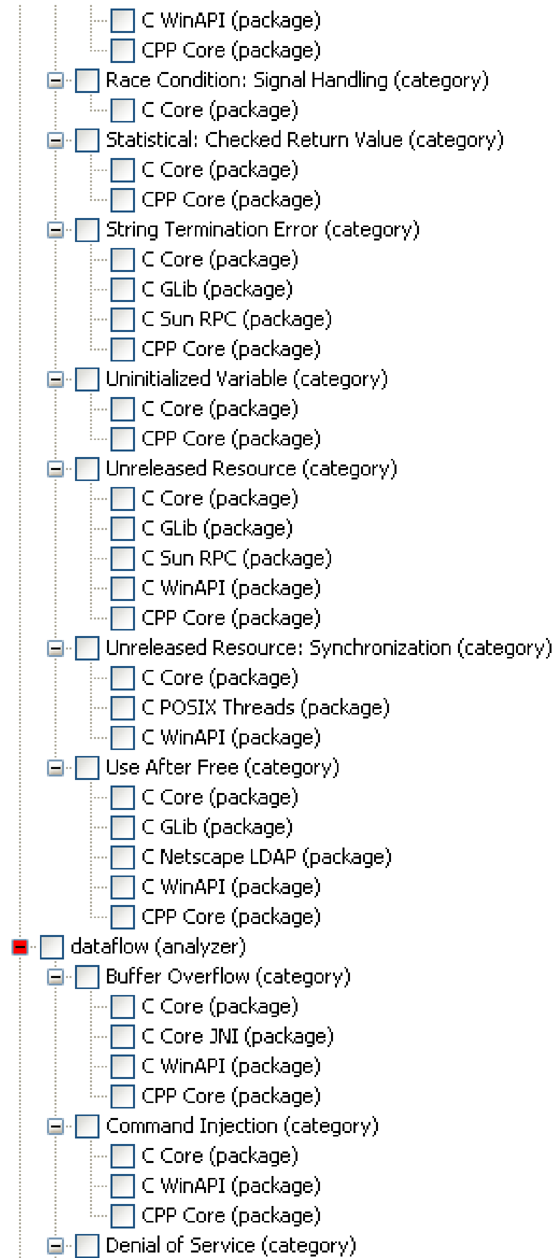
A

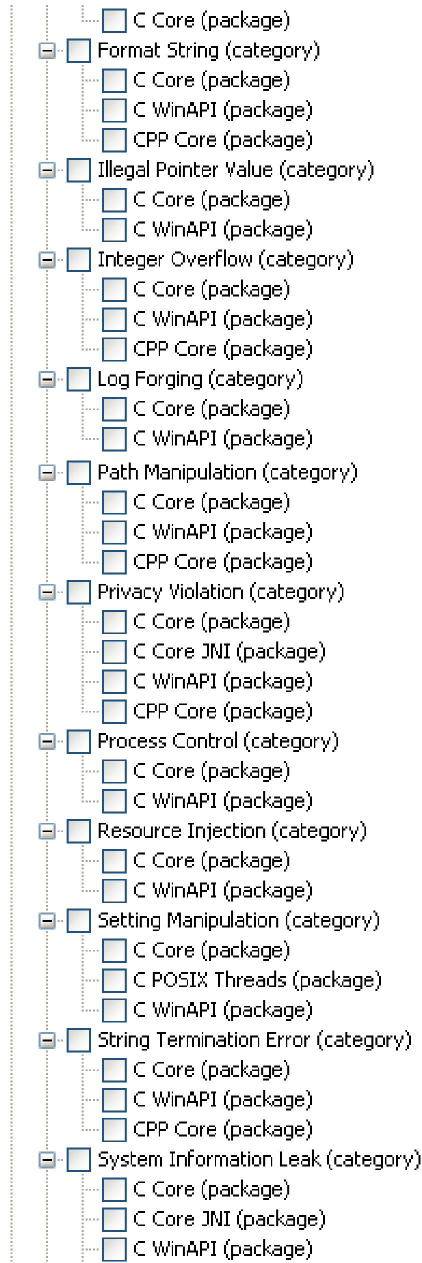
Firme di rilevamento del software Fortify relative alle vulnerabilità riscontrabili nei codici sorgente C/C++

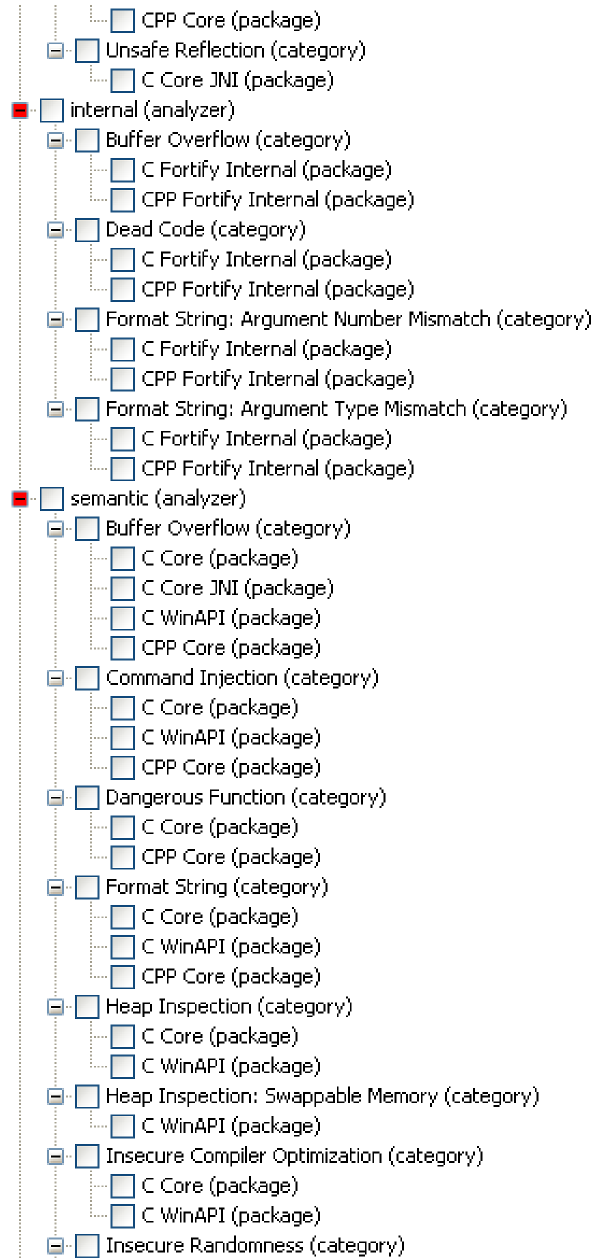
La presente Appendice illustra le firme di rilevamento utilizzate dal software Fortify all'interno del Rulepack Management per i linguaggi di programmazione C/C++; esse sono racchiuse nel pacchetto identificato con il nome "Fortify Secure Coding Rules, Core, C/C++". Per ciascuna firma viene dapprima evidenziato l'analizzatore utilizzato; successivamente vengono considerati la categoria e il package di appartenenza.

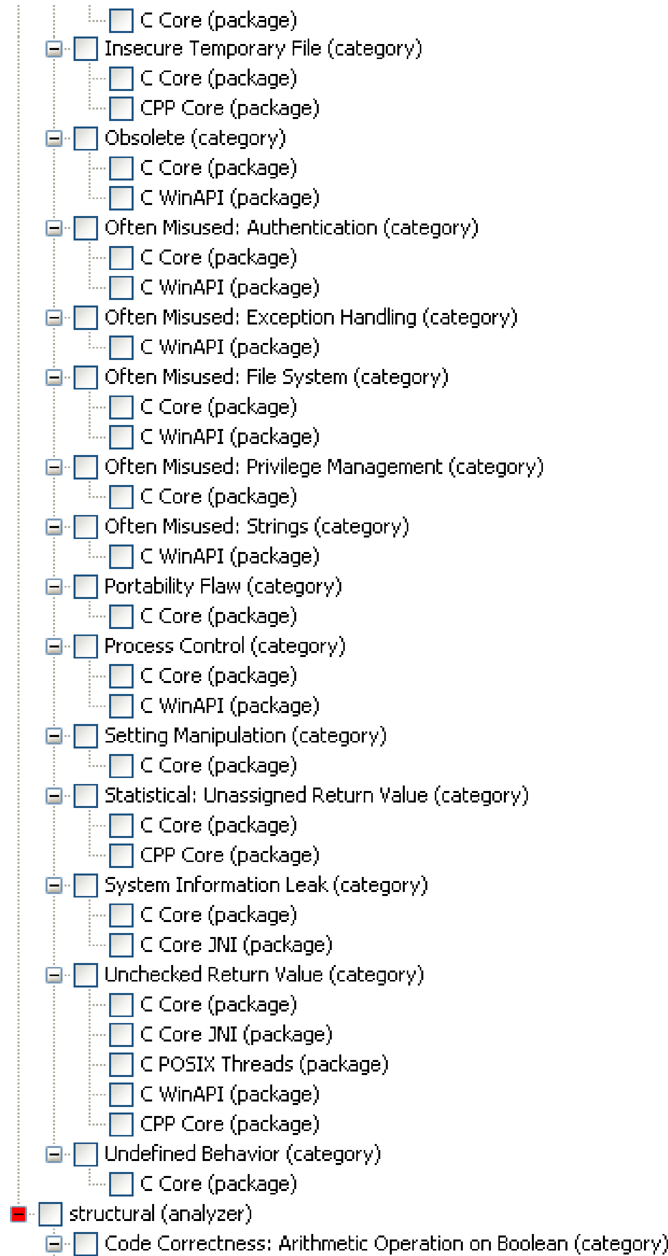


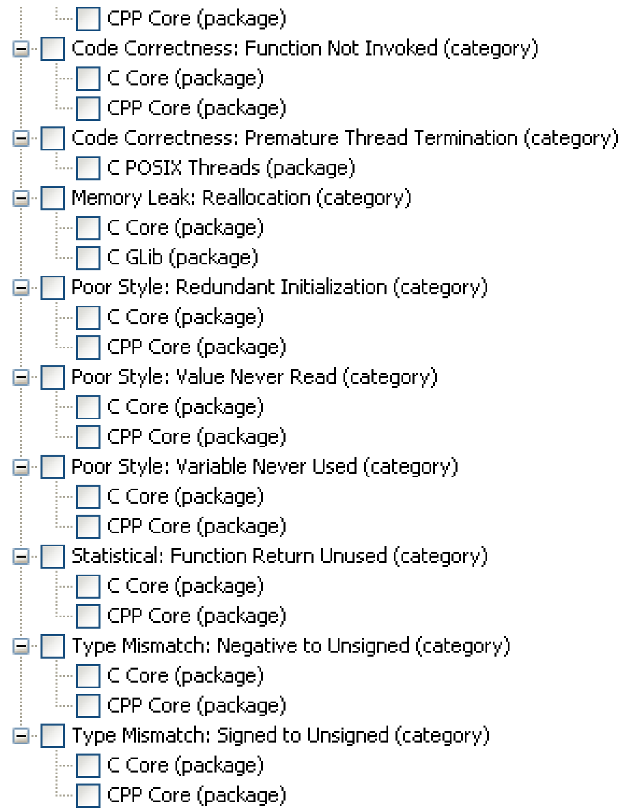

CPP Core (package)
Double Free (category)
C Core (package)
C GLib (package)
C Netscape LDAP (package)
C WinAPI (package)
CPP Core (package)
Insecure Temporary File (category)
C Core (package)
C GLib (package)
Least Privilege Violation (category)
C Core (package)
C GLib (package)
CPP Core (package)
Memory Leak (category)
C Core (package)
C GLib (package)
C Netscape LDAP (package)
C Sun RPC (package)
C WinAPI (package)
CPP Core (package)
Memory Leak: Reallocation (category)
C Core (package)
C GLib (package)
C Netscape LDAP (package)
C WinAPI (package)
CPP Core (package)
Missing Check against Null (category)
C ActiveDirectory LDAP (package)
C Core (package)
C GLib (package)
C Netscape LDAP (package)
C Sun RPC (package)
C WinAPI (package)
CPP Core (package)
Null Dereference (category)
C Core (package)
CPP Core (package)
Race Condition: File System Access (category)
C Core (package)
C GLib (package)
C Netscape LDAP (package)


C WinAPI (package)
CPP Core (package)
Race Condition: Signal Handling (category)
C Core (package)
Statistical: Checked Return Value (category)
C Core (package)
CPP Core (package)
String Termination Error (category)
C Core (package)
C GLib (package)
C Sun RPC (package)
CPP Core (package)
Uninitialized Variable (category)
C Core (package)
CPP Core (package)
Unreleased Resource (category)
C Core (package)
C GLib (package)
C Sun RPC (package)
C WinAPI (package)
CPP Core (package)
Unreleased Resource: Synchronization (category)
C Core (package)
C POSIX Threads (package)
C WinAPI (package)
Use After Free (category)
C Core (package)
C GLib (package)
C Netscape LDAP (package)
C WinAPI (package)
CPP Core (package)
dataflow (analyzer)
Buffer Overflow (category)
C Core (package)
C Core JNI (package)
C WinAPI (package)
CPP Core (package)
Command Injection (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Denial of Service (category)


C Core (package)
Format String (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Illegal Pointer Value (category)
C Core (package)
C WinAPI (package)
Integer Overflow (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Log Forging (category)
C Core (package)
C WinAPI (package)
Path Manipulation (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Privacy Violation (category)
C Core (package)
C Core JNI (package)
C WinAPI (package)
CPP Core (package)
Process Control (category)
C Core (package)
C WinAPI (package)
Resource Injection (category)
C Core (package)
C WinAPI (package)
Setting Manipulation (category)
C Core (package)
C POSIX Threads (package)
C WinAPI (package)
String Termination Error (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
System Information Leak (category)
C Core (package)
C Core JNI (package)
C WinAPI (package)


CPP Core (package)
Unsafe Reflection (category)
C Core JNI (package)
internal (analyzer)
Buffer Overflow (category)
C Fortify Internal (package)
CPP Fortify Internal (package)
Dead Code (category)
C Fortify Internal (package)
CPP Fortify Internal (package)
Format String: Argument Number Mismatch (category)
C Fortify Internal (package)
CPP Fortify Internal (package)
Format String: Argument Type Mismatch (category)
C Fortify Internal (package)
CPP Fortify Internal (package)
semantic (analyzer)
Buffer Overflow (category)
C Core (package)
C Core JNI (package)
C WinAPI (package)
CPP Core (package)
Command Injection (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Dangerous Function (category)
C Core (package)
CPP Core (package)
Format String (category)
C Core (package)
C WinAPI (package)
CPP Core (package)
Heap Inspection (category)
C Core (package)
C WinAPI (package)
Heap Inspection: Swappable Memory (category)
C WinAPI (package)
Insecure Compiler Optimization (category)
C Core (package)
C WinAPI (package)
Insecure Randomness (category)


C Core (package)
Insecure Temporary File (category)
C Core (package)
CPP Core (package)
Obsolete (category)
C Core (package)
C WinAPI (package)
Often Misused: Authentication (category)
C Core (package)
C WinAPI (package)
Often Misused: Exception Handling (category)
C WinAPI (package)
Often Misused: File System (category)
C Core (package)
C WinAPI (package)
Often Misused: Privilege Management (category)
C Core (package)
Often Misused: Strings (category)
C WinAPI (package)
Portability Flaw (category)
C Core (package)
Process Control (category)
C Core (package)
C WinAPI (package)
Setting Manipulation (category)
C Core (package)
Statistical: Unassigned Return Value (category)
C Core (package)
CPP Core (package)
System Information Leak (category)
C Core (package)
C Core JNI (package)
Unchecked Return Value (category)
C Core (package)
C Core JNI (package)
C POSIX Threads (package)
C WinAPI (package)
CPP Core (package)
Undefined Behavior (category)
C Core (package)
structural (analyzer)
Code Correctness: Arithmetic Operation on Boolean (category)


CPP Core (package)
Code Correctness: Function Not Invoked (category)
C Core (package)
CPP Core (package)
Code Correctness: Premature Thread Termination (category)
C POSIX Threads (package)
Memory Leak: Reallocation (category)
C Core (package)
C GLib (package)
Poor Style: Redundant Initialization (category)
C Core (package)
CPP Core (package)
Poor Style: Value Never Read (category)
C Core (package)
CPP Core (package)
Poor Style: Variable Never Used (category)
C Core (package)
CPP Core (package)
Statistical: Function Return Unused (category)
C Core (package)
CPP Core (package)
Type Mismatch: Negative to Unsigned (category)
C Core (package)
CPP Core (package)
Type Mismatch: Signed to Unsigned (category)
C Core (package)
CPP Core (package)

B

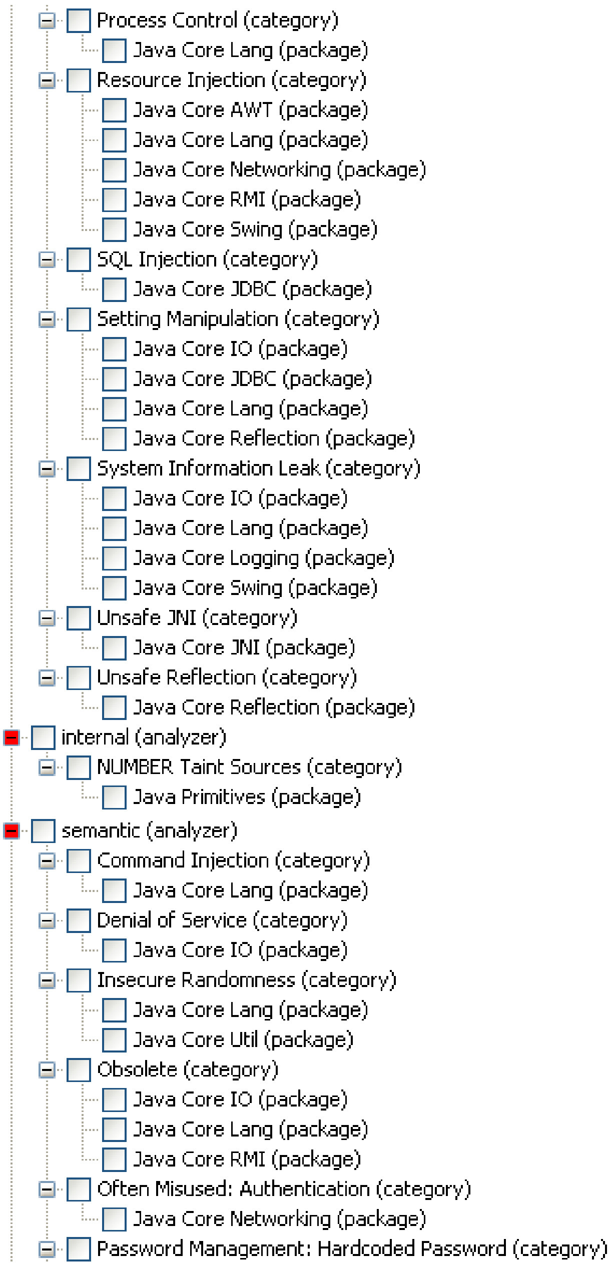
Firme di rilevamento del software Fortify relative alle vulnerabilità riscontrabili nei codici sorgente Java

La presente Appendice illustra le firme di rilevamento utilizzate dal software Fortify all'interno del Rulepack Management per il linguaggio di programmazione Java; esse sono racchiuse nel pacchetto identificato con il nome "Fortify Secure Coding Rules, Core, Java". Per ciascuna firma viene dapprima evidenziato l'analizzatore utilizzato; successivamente vengono considerati la categoria e il package di appartenenza.

- Command Line Arguments (inputsource)
- Database (inputsource)
- Environment Variables (inputsource)
- GUI Form (inputsource)
- Java Properties (inputsource)
- Private Information (inputsource)
- Serialized Data (inputsource)
- Stream (inputsource)
- System Information (inputsource)
- Web (inputsource)
- XML Document (inputsource)
- controlflow (analyzer)
  - Missing Check against Null (category)
    - Java Apache Struts (package)
    - Java Core IO (package)
    - Java Core Lang (package)
    - Java Hibernate (package)
    - Java J2EE ServletAPI (package)
    - Java Netscape LDAP (package)
    - Java Spring Beans (package)
    - Java Spring Hibernate (package)

- Java Spring Hibernate3 (package)
- Missing Check for Null Parameter (category)
  - Java Core Lang (package)
- Null Dereference (category)
  - Java Core Lang (package)
- Statistical: Checked Return Value (category)
  - Java Core Lang (package)
- Unreleased Resource: Database (category)
  - Java Core JDBC (package)
  - Java Hibernate (package)
- Unreleased Resource: Streams (category)
  - Java Core IO (package)
  - Java Netscape LDAP (package)

- dataflow (analyzer)
  - Access Control: Database (category)
    - Java Core JDBC (package)
  - Code Correctness: Erroneous Class Compare (category)
    - Java Core Lang (package)
  - Command Injection (category)
    - Java Core IO (package)
    - Java Core Lang (package)
  - Cross-Site Scripting (category)
    - Java Core IO (package)
  - Denial of Service (category)
    - Java Core IO (package)
    - Java Core Lang (package)
  - JavaScript Hijacking: Ad Hoc Ajax (category)
    - Java Core IO (package)
  - Log Forging (category)
    - Java Core Logging (package)
  - Password Management (category)
    - Java Core JDBC (package)
  - Password Management: Weak Cryptography (category)
    - Java Core JDBC (package)
  - Path Manipulation (category)
    - Java Core IO (package)
    - Java Core Util (package)
  - Privacy Violation (category)
    - Java Core IO (package)
    - Java Core Lang (package)
    - Java Core Logging (package)
    - Java Core Swing (package)


Process Control (category)
Java Core Lang (package)
Resource Injection (category)
Java Core AWT (package)
Java Core Lang (package)
Java Core Networking (package)
Java Core RMI (package)
Java Core Swing (package)
SQL Injection (category)
Java Core JDBC (package)
Setting Manipulation (category)
Java Core IO (package)
Java Core JDBC (package)
Java Core Lang (package)
Java Core Reflection (package)
System Information Leak (category)
Java Core IO (package)
Java Core Lang (package)
Java Core Logging (package)
Java Core Swing (package)
Unsafe JNI (category)
Java Core JNI (package)
Unsafe Reflection (category)
Java Core Reflection (package)
internal (analyzer)
NUMBER Taint Sources (category)
Java Primitives (package)
semantic (analyzer)
Command Injection (category)
Java Core Lang (package)
Denial of Service (category)
Java Core IO (package)
Insecure Randomness (category)
Java Core Lang (package)
Java Core Util (package)
Obsolete (category)
Java Core IO (package)
Java Core Lang (package)
Java Core RMI (package)
Often Misused: Authentication (category)
Java Core Networking (package)
Password Management: Hardcoded Password (category)

- Java Core JDBC (package)
- Process Control (category)
  - Java Core Lang (package)
- SQL Injection (category)
  - Java Core JDBC (package)
- Statistical: Unassigned Return Value (category)
  - Java Core Lang (package)
- System Information Leak (category)
  - Java Core Lang (package)
- Unchecked Return Value (category)
  - Java Core Beans (package)
  - Java Core IO (package)
  - Java Core JDBC (package)
  - Java Core Lang (package)
  - Java Core NIO (package)
  - Java Core RMI (package)
  - Java Core Reference (package)
  - Java Core Security (package)
- Unsafe JNI (category)
  - Java Core JNI (package)
- structural (analyzer)
  - Poor Style: Redundant Initialization (category)
    - Java Core Lang (package)
  - Poor Style: Value Never Read (category)
    - Java Core Lang (package)
  - Statistical: Function Return Unused (category)
    - Java Core Lang (package)

Ringraziamenti

Dedico il presente lavoro di tesi a tutte le persone che hanno condiviso con me questi anni intensi e impegnativi.

Il primo "grazie" va alla mia famiglia che mi è sempre stata vicina dimostrandomi il suo affetto.

Un ringraziamento speciale è per l'Amico Prof. Domenico Ursino che ha rappresentato per me un costante punto di riferimento; la passione e la pazienza che mette nel suo lavoro e la disponibilità nei confronti di ogni studente sono doti che non finirò mai di apprezzare abbastanza.

"Grazie" agli Amici e ai Colleghi con i quali ho condiviso i successi e le difficoltà della vita universitaria; li ringrazio soprattutto per i bei momenti trascorsi assieme al di fuori dell'università.

Ringrazio Salvatore Pullano e Francesco Profazio, che mi hanno introdotto, con grande capacità, in una realtà aziendale complessa quale è quella della sicurezza informatica, accogliendomi benevolmente nella "Capitale".

Ringrazio ancora il Dott. Massimo Proietti per la fiducia che ha riposto in me, dandomi l'opportunità di confrontarmi con una realtà importante e permettendomi di fare un'esperienza umana e professionale certamente significativa e fruttuosa.