POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea

Analisi statica per l'identificazione di vulnerabilità in ambiente J2EE

Relatore:

Prof. Giuseppe Serazzi

Correlatore:

Ing. Stefano Zanero

Laureando:

Luca Carettoni - Matr. 667031

Anno Accademico 2005–2006

Just Thinking Outside The Box

Indice

Capitolo 1

Introduzione

1.1 Contesto

Con la progressiva diffusione di architetture distribuite, aperte e flessibili, garantire la sicurezza e l'integrità dei sistemi informativi aziendali è diventato un compito complesso; se da un lato le applicazioni web hanno portato evidenti benefici in termini di fruibilità per l'utente, dall'altro hanno sicuramente introdotto un nuovo elemento debole ai sistemi. Nella moderna "società della conoscenza" aziende e privati si affidano sempre più a soluzioni software basate su tecnologie web che eliminano definitivamente le problematiche di accesso alle risorse in termini spaziali e temporali.

Per loro natura i servizi in Rete sono forniti in maniera aperta verso utenti generici, ricevendo e gestendo informazioni su un canale tendenzialmente inaffidabile. In questa fase di cambiamento delle tecnologie informatiche, di apertura verso la condivisione delle informazioni, ma anche di attenzione verso la protezione della conoscenza, le metodologie e le tecniche per la sicurezza informatica rivestono un ruolo importante. Durante lo sviluppo di software è necessario progettare e realizzare infrastrutture informatiche che, oltre a svolgere le funzioni per le quali sono preposte, consentano di soddisfare i requisiti di confidenzialità delle informazioni, integrità dei dati e disponibilità del servizio. I linguaggi di programmazione moderni offrono sempre più possibilità e flessibilità a chi crea un applicativo, ma contempo-

raneamente esigono una crescente attenzione da parte del programmatore, che può incorrere in sviste e ingenuità, dimenticando la validazione ed il controllo di certi dati utilizzati in situazioni potenzialmente rischiose. Queste disattenzioni non vanno a minare il corretto funzionamento dell'applicativo così come è stato progettato, ma non precludono variazioni non autorizzate del flusso di esecuzione delle istruzioni, nè tantomeno l'esposizione di dati riservati.

Per meglio formalizzare la definizione di vulnerabilità software possiamo dire che un programma vulnerabile è un insieme di istruzioni che, pur rispettando le regole sintattiche del linguaggio stesso, permette alterazioni del flusso di esecuzione che conducono ad una deviazione dal comportamento atteso e alla fuoriuscita di informazioni private. Per ridurre al minimo queste possibili deviazioni indesiderate, gli sviluppatori devono essere in grado di progettare ed implementare applicazioni sicure, ma anche di analizzare le applicazioni scritte al fine di individuare quelle vulnerabilità che, durante lo sviluppo del software, posso essere state introdotte erroneamente.

1.2 Obiettivi della tesi

Il lavoro di tesi verte intorno al problema dell'individuazione di vulnerabilità attraverso l'analisi statica del codice sorgente. Questo genere di analisi rappresenta una delle possibili soluzioni per la ricerca automatica di problematiche di sicurezza all'interno delle applicazioni; sebbene lo studio di queste tecniche risalga ormai agli albori dell'informatica, l'applicazione in ambito web rappresenta una nuova sfida per gli esperti ed i ricercatori. Siamo ben consapevoli dell'impossibilità teorica di identificare, con un'analisi di questo tipo, tutte le differenti tipologie di vulnerabilità; a questo proposito abbiamo preferito concentrate la nostra attenzione verso l'ampia categoria delle *Unvalidated Input Flaw* ed in particolare di concentrarci sull'ambiente J2EE. Sebbene gran parte dei principi siano di applicabilità generale, il caso specifico illustrato fa riferimento al mondo delle applicazioni web.

Lo studio intrapreso è volto a proporre una metodologia di analisi statica basata sulla ricostruzione del valore delle variabili stringhe all'interno dei metodi considerati pericolosi, al fine di rispondere alla semplice domanda: "Quali possibili valori assumerà questa variabile di tipo stringa, in questo

punto del codice, durante l'esecuzione?". Identificando metodi pericolosi e potenziale valore delle variabili di tipo stringa, risulta possibile comparare ogni invocazione con le regole per considerarla esente da problematiche di sicurezza. Negli obiettivi di questo lavoro di tesi rientrano la definizione formale del metodo proposto, l'implementazione di uno strumento software e la valutazione dell'efficacia attraverso l'analisi su software campione e security benchmark.

Per la natura dell'applicazione, si pongono naturalmente una serie di requisiti che il metodo e la sua implementazione dovranno rispettare.

Riconoscimento: banalmente, deve essere possibile riconoscere punti vulnerabili delle applicazioni web analizzate, a dispetto dei diversi fattori di approssimazione.

Analisi statica: occorre progettare un sistema che operi *offline*, ovvero analizzando il solo codice sorgente, senza interloquire con il server che andrà ad ospitare l'applicazione e senza eseguire l'applicazione stessa.

Codice sorgente: deve essere disponibile tutto il codice chiamato in causa durante l'esecuzione, pena una perdita di precisione (anche sensibile) dei dati forniti dall'analisi: vogliamo realizzare uno strumento di *supporto al programmatore*, che quindi dispone del *source code* che ha scritto. Nel caso di JSP/JAVA il *bytecode* viene considerato come una forma alternativa di codice in quanto forma intermedia di linguaggio interpretabile.

Espandibilità: date le differenti tecnologie di *server-side scripting* occorre portare avanti un progetto che sia il più possibile aperto a evoluzioni ed espansioni, nel senso di incremento di linguaggi riconoscibili con le loro peculiarità e astrazioni.

Aggiornabilità: spostando l'attenzione sull'oggetto della ricerca, le vulnerabilità del codice appunto, il sistema deve essere facilmente modificabile al fine di individuare nuove tipologie o nuove occorrenze di quelle note.

Reporting: al termine della scansione, deve essere possibile redigere un

rapporto circa lo stato del codice analizzato e le eventuali vulnerabilità rilevate.

1.3 Sviluppo del lavoro

Il percorso di studio, che ha portato alla creazione di uno strumento software per l'identificazione automatica di vulnerabilità nelle applicazioni web, si è così sviluppato:

- Studio del contesto applicativo e delle tecnologie utilizzate per implementare soluzioni applicative in rete; enumerazione delle vulnerabilità in ambiente web; identificazione delle problematiche di sicurezza che risultano analizzabili, in maniera automatica, attraverso analisi statica sul codice sorgente.
- Studio e formalizzazione di una metodologia di analisi che consideri le variabili di tipo stringa e valuti sistematicamente ogni singola operazione, presente nel codice sorgente, su quella tipologia.
- Implementazione di uno strumento software che utilizzi lo studio precedentemente realizzato, al fine di identificare problematiche di sicurezza nel software. Tale applicativo è rivolto a sviluppatori e tester che intendono monitorare il proprio software durante tutte le fasi del processo di sviluppo.
- Testing su applicazioni campione, su progetti reali e/o security benchmark; valutazione del metodo proposto; comparazione rispetto ad altri strumenti che utilizzino sempre un processo di revisione statica per l'identificazione delle falle.

1.4 Struttura del documento di tesi

Il presente documento è così strutturato:

- Il **Capitolo 2** si occupa della definizione dello spazio del problema: partendo dai concetti generali e dalle tecnologie utilizzate oggigiorno per lo sviluppo di applicativi in rete, si arriva a specificare il reale

ambito di applicabilità del metodo, illustrando le tipologie di vulnerabilità che possono essere trattate. Attraverso considerazioni empiriche vengono valutate le soluzioni esistenti cercando di illustrare pregi e difetti dei diversi approcci.

- Nel **Capitolo 3** si descrive il processo adottato per la costruzione dei linguaggi facendo corrispondere la divisione logica delle fasi dell'analisi allo svolgimento del capitolo. Illustreremo in queste pagine la definizione di *Flow Graph* del programma in analisi, la costruzione della grammatica Context-Free associata e la sua approssimazione ad automa a stati finiti.

- Nel **Capitolo 4** si espone il metodo di comparazione dei linguaggi *safe* ed *unsafe* associati ad ogni invocazione ritenuta potenzialmente pericolosa. In particolare si definiranno formalmente gli algoritmi e le strategie adottate per rappresentare e comparare i linguaggi, oltre alle approssimazioni introdotte. Infine, si introdurranno le problematiche e le possibili soluzioni associate a casistiche che possono generare falsi negativi e positivi.

- Nel **Capitolo 5** si riassume il design e l'implementazione dello strumento software utilizzato per validare scientificamente la metodologia di analisi proposta. Tramite la sperimentazione su software reale si individua l'applicabilità della soluzione e se ne verifica l'impatto sul ciclo di sviluppo. In questo capitolo, ci si sofferma anche su riflessioni in merito all'espandibilità dell'architettura e alle possibili evoluzioni del software stesso, dedicando spazio a brevi digressioni implementative.

- Il **Capitolo 6** conclude la trattazione esponendo i risultati raggiunti ed i possibili sviluppi futuri del metodo.

Capitolo 2

Applicazioni web: vulnerabilità e rischi

Sebbene il metodo proposto ed i concetti contenuti nel lavoro di tesi possono essere ritenuti di carattere generale, si è deciso di restringere il campo di indagine al mondo delle applicazioni web (d'ora in poi nominate con l'acronimo **WA**). Il lettore interessato potrà facilmente portare molti dei concetti affrontati anche nel caso di analisi statica su applicativi Java stand-alone.

Una WA può essere definita come un programma, sviluppato al fine di svolgere determinate funzioni, che utilizza tecnologie Internet per fornire contenuti e risorse verso utenti che vi accedono tramite un normale *web browser*. Tecnicamente parlando però una WA può essere un'entità di notevole complessità: codice eseguibile presente nei web server e negli application server, differenti tecnologie di presentazione ed elaborazione delle informazioni, database, interfacce verso sistemi legacy, etc. L'impatto delle WA nel mondo di Internet si è reso sempre più deciso ed evidente negli ultimi anni quando le tecnologie web-oriented e i linguaggi di programmazione o di descrizione (sempre più di alto livello) sono diventati alla portata ormai di tutti. Queste innovazioni a livello tecnologico hanno suscitato un corrispettivo interesse nei più disparati settori economici: quasi tutte le aziende vogliono essere presenti su Internet offrendo informazioni e servizi ai propri clienti [1]. Secondo le recenti stime dell'Internet World Stats [2] il 16.7%

dell'intera popolazione mondiale utilizza Internet e l'incremento dal 2000 al 2006 è pari 200.9%; questi dati dimostrano inequivocabilmente come le tecnologie web, i servizi correlati e le potenziali problematiche di sicurezza siano argomenti di estrema attualità e interesse sociale.

2.1 Tecnologie

Lo scopo delle WA è quello di fornire interattività e funzionalità all'utente, per scopi diversi a seconda delle finalità per cui la stessa applicazione è stata progettata (ad esempio: esperienza di navigazione in Internet fine a sè stessa, operazioni bancarie, interrogazioni di basi di dati distribuite...). Il fattore che accomuna le differenti situazioni proposte risiede nella generazione dei dati (pagine HTML nella fattispecie) che giungono all'utente in seguito a una richiesta: la generazione dei contenuti è sempre *dinamica*. L'interazione dell'utente con l'applicazione avviene attraverso uno scambio di messaggi, secondo il protocollo HTTP [3], su un canale di trasmissione fornito dal protocollo TCP/IP [4, 5].

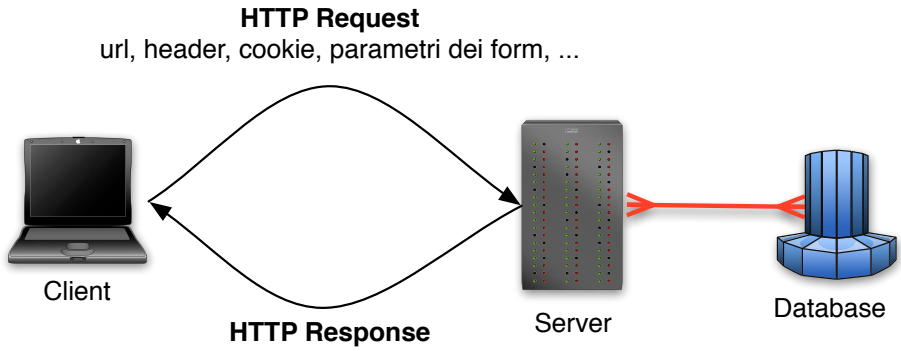

Figura 2.1: Il normale flusso di comunicazione tra client e web server

Questo significa che il dato richiesto dall'utente non è già disponibile in un formato completo e "impacchettato", ma deve essere costruito istantaneamente dal server che gestisce la WA interagendo con i componenti di rete ad esso correlati. Il processo descritto avviene se la logica applicativa risiede sul lato server della connessione che permette l'accesso all'applicazio-

ne; in questo caso l'esecuzione o l'interpretazione dei comandi (in generale degli *script*) è di tipo *server-side*, ed è proprio il server a dover disporre di determinati componenti per svolgere queste operazioni.

Tra le tecnologie operanti in questo ambito ricordiamo *JSP (Java Server Pages)* di Sun Microsystems [6], *PHP (PHP Hypertext Processor)* di licenza pubblica [7], e *ASP (Active Server Pages)* di Microsoft [8]. Deve essere inoltre citata la nota interfaccia *CGI* (Common Gateway Interface), che però si differenzia dalle precedenti soluzioni poichè costituisce unicamente un protocollo per l'invocazione da parte del server di programmi dedicati. I paradigmi citati in precedenza invece sono esempi di *scripting-language* lato server, e sono caratterizzati dall'immersione del codice eseguibile all'interno di una normale sintassi HTML. Tutti questi sistemi sono accomunati da caratteristiche di potenzialità e usabilità analoghe, e come detto, dal tipo di ripartizione della logica applicativa, che risiede interamente sul lato server. Paradigmi più semplici di applicazioni web prevedono invece l'interpretazione degli script su lato client (trasferendo ad esso il carico della logica dell'applicazione), tramite plug-in del browser (per esempio *Javascript* di NetScape oppure l'evoluzione asincrona *AJAX*): la logica applicativa lato client comporta una minore libertà nella creazione di applicativi, e inoltre aumenta i rischi di attacco al server, poichè l'utente, una volta bypassati i controlli locali, ha la strada libera verso il cuore del sistema informativo remoto. La strategia intermedia, come terza ipotesi, prevede la ripartizione della logica applicativa tra lato server e lato client, dotando il client di funzionalità limitate e di supporto, e riservando comunque al server i compiti più critici in termini di complessità computazionale ma anche sicurezza.

Questo lavoro di tesi analizzerà in dettaglio le problematiche di sicurezza relative ai sistemi basati su architettura J2EE, con tecnologie JSP o in generale Servlet.

Sun Microsystems ha definito con il termine "Enterprise Computing" un nuovo paradigma di calcolo distribuito eseguito attraverso un gruppo di programmi interagenti attraverso la rete. In questa architettura, ogni componente utilizza diversi protocolli di rete e altrettanti standard per l'elaborazione e la presentazione dei dati. La soluzione proposta da Sun cerca di semplificare l'intrinseca complessità di tali sistemi, fornendo un ambien-

te di sviluppo e fruizione, completo di API e metodologie di approccio al problem solving [9]; questa soluzione è composta da quattro elementi principali: Specifics, Reference Implementation, Compatibility Test, Application Programming Model. Le specifiche elencano gli elementi necessari alla piattaforma e le procedure da eseguire per una corretta implementazione con J2EE. La Refence Implementation contiene prototipi che rappresentano istanze semanticamente corrette di implementazioni J2EE al fine di fornire all'industria del software modelli completi per il test; include anche tool per il deployment e l'amministrazione di sistema, EJB, JSP, container per il supporto a runtime, Java Messaging Service e altri prodotti di terzi. L'Application Programming Model è un modello per la progettazione e la programmazione di applicazioni basato su best-practice, per favorire un approccio ottimale alla piattaforma.

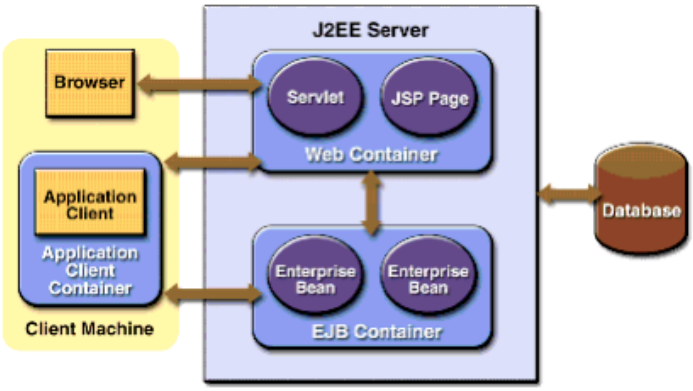

Figura 2.2: Servlet e JSP come elementi del Web-Tier

L'architettura proposta dalla piattaforma J2EE divide le applicazioni enterprise in tre stati applicativi fondamentali: componenti, contenitori e connettori. Il modello di programmazione prevede lo sviluppo di soluzioni utilizzando componenti che posso essere classificati all'interno delle seguenti tecnologie:

- Enterprise Java Bean (EJB)
- Servlet

- JSP, Java Server Page
- Applet Java

Il primo fornisce un supporto per la creazione di componenti served-side che possono essere generati indipendentemente dalla specifica base di dati, da uno specifico transaction server o dalle diverse piattaforme di fruizione. Le servlet consentono la costruzione di servizi web altamente performanti ed in grado di funzionare sulla maggior parte dei web server presenti oggi sul mercato. La terza tecnologia, le Java Server Page permettono la costruzione di pagine HTML dai contenuti dinamici utilizzando tutta la potenza del linguaggio Java. Le Applet rappresentano invece la risposta di Sun alla logica applicativa lato client.

Senza addentrarci nella descrizione dettagliata della tecnologia JSP, per le quali si rimanda a [10, 11, 12], è utile ricordare alcune caratteristiche principali che sarà opportuno tenere in considerazione durante il processo di sviluppo del nostro strumento di analisi automatica. Sebbene il metodo di analisi basato sulla ricostruzione del valore delle variabili di tipo stringa possa essere utilizzato per qualsiasi linguaggio server-side, l'implementazione del tool attuale si rifà proprio a questo *scripting language*.

JSP possiede peculiarità che lo rendono molto appetibile in un'ottica di ampio respiro:

- è largamente utilizzato e la sua diffusione è legata a quella del "progenitore" Java; presumibilmente sempre più WA saranno scritte in JSP;
- è dotato di astrazioni molto comode che permettono di stabilire chiaramente il punto di ingresso dei dati all'interno della WA (metodi di tipo *getter* sull'oggetto *request*);
- la sua architettura prevede la traduzione della pagina contenente script *embedded* in una servlet (che è 100% *pure Java*), e quindi è possibile trasferire il problema dall'analisi di un file JSP all'analisi del relativo codice Java (rendendo di fatto il nostro metodo aperto a una implementazione indipendente dal contesto web).

Come appena anticipato, il meccanismo fondamentale che muove la tecnologia JSP è la traduzione in servlet. Queste sono classi Java particolari che mettono a disposizione le funzionalità del web server, e dispongono di una serie di strutture accessorie atte ad agevolare l'interconnessione e l'operatività remota; sono necessarie per lo scambio di dati o, ad esempio, per la gestione della sessione di un utente connesso (ricordiamo, infatti, che il protocollo HTTP è *stateless*). Un file JSP può contenere comandi, le *direttive*, che rappresentano determinate azioni tradotte in seguito nel codice Java della rispettiva servlet: ogni file *.jsp* viene trasposto in servlet e compilato la prima volta che viene richiesto, causando un minimo overhead per il server. Ma alle successive invocazioni della stessa pagina, la servlet già compilata riceverà la richiesta, consentendo una rapida interpretazione del suo contenuto. Il ciclo di vita di una servlet [10] prevede il servizio di tutte le richieste in arrivo fino alla distruzione della servlet stessa ad opera del server: metodi appositi servono le richieste HTTP di diverso tipo, ed è possibile integrare servlet "native" con pagine JSP per ottenere una maggiore divisione della logica dell'applicazione (servlet) dalla presentazione dei dati (JSP).

Container

Conosciamo il meccanismo tramite il quale ogni pagina JSP viene convertita in una servlet che ne rispecchi le direttive, ma manca ancora il punto di interconnessione tra il mondo delle servlet ed il web server. Questo "trait d'union" è rappresentato dal *container*, l'ambiente di runtime di ogni servlet presente nel sistema. In qualità di ambiente ospitante le classi Java, il container ha il compito di inoltrare le richieste (messaggi *request*) alle servlet corrette, dopo averle selezionate in seguito alle richieste HTTP; il container *filtra* ogni messaggio in transito e attiva le risorse appropriate. Lo stesso comportamento si ha in fase di risposta, attraverso l'interpretazione dell'oggetto *response* e la produzione del messaggio HTTP da restituire al client remoto.

I container possono essere di tipo *embedded* (incorporati nella struttura nativa del server ospitante) o *add-on*, ovvero moduli aggiuntivi che rendono *servlet-enabled* un web server specifico; gerarchicamente inoltre, esistono

altri ambienti contenuti all'interno di un container, che raggruppano servlet di una medesima WA.

Oggetti request e response

Questi due oggetti particolari sono presenti come default in ogni servlet, e il metodo di servizio li possiede in qualità di argomenti (sono perciò variabili liberamente referenziabili all'interno del suo corpo). Mentre il primo ospita i dati legati al messaggio di richiesta inviato dal client, il secondo contiene le informazioni che dovranno essere inserite nella risposta. In particolare, l'oggetto request ospita tutti i parametri della richiesta, comprese le stringhe eventualmente immesse dall'utente remoto in form apposite: interrogando questo oggetto, è possibile ottenere in formato "object-oriented" tutto quello che contraddistingue il messaggio HTTP sottomesso dal client connesso al web server. Proprio l'oggetto *request* costituisce un punto fondamentale per la nostra analisi poichè rappresenta il principale punto di ingresso dei dati esterni che il programmatore dovrà avere l'accortezza di validare.

Propagazione e interruzione delle richieste

Solitamente un'operazione svolta all'interno di una applicazione pensata per il web porta alla creazione di contenuti appositi a partire da determinate richieste di provenienza remota. Ma non è detto che il ciclo *richiesta - computazione - risposta* preveda l'interrogazione di un'unica risorsa: in generale, sarà possibile che la prima pagina contattata dia il via ad una *catena* di elaborazioni separate, soprattutto se la WA è scritta in base a principi di modularità e divisione di logica applicativa e presentazione.

Una pagina JSP (e quindi una servlet) può passare il controllo ad un'altra in differenti modi:

- attraverso direttiva di *include*: in questo modo la servlet generata contiene già il codice relativo alla pagina JSP chiamante e a quella invocata;
- mediante direttiva di *forward*: in questo caso il controllo è letteralmente trasferito ad una nuova pagina JSP (servlet), formattando una

richiesta con lo stesso oggetto *request* attualmente processato, al quale è eventualmente possibile aggiungere nuovi parametri;

- tramite redirezione: è una funzionalità che causa l'invio della risposta al client, che viene forzato ad effettuare una nuova richiesta. Concettualmente non è un caso interessante, poichè riassume la situazione di due richieste successive e indipendenti;
- attraverso comportamento reattivo agli errori: una risorsa JSP può essere configurata per effettuare una sorta di *forward* ad una pagina di gestione degli errori in caso di eccezioni riscontrate a runtime; concettualmente questa politica equivale ad inoltrare la richiesta alla nuova pagina.

Dopo aver introdotto le caratteristiche dell'ambiente J2EE ed aver compreso le differenti modalità di esecuzione e di utilizzo dei componenti dell'architettura è facile comprendere perchè le WA rappresentano uno dei più attuali ambiti d'azione per lo sfruttamento di vulnerabilità del codice. La lotta contro questo fenomeno è già il principale campo di ricerca di numerose società private: esistono altresì progetti di tipo Open Source che studiano contromisure e diffondono conoscenza sul tema delle vulnerabilità, il più importante dei quali è senza dubbio OWASP (*Open Web Application Security Project*), dal quale derivano idee e soluzioni pratiche, nonchè una guida alla costruzione di WA sicure [13] e una top-ten delle vulnerabilità più diffuse [14], categorizzate per severità e rischio. Proprio attraverso questo documento forniremo una prima classificazione delle vulnerabilità nelle WA.

2.2 Vulnerabilità

È ormai opinione diffusa [15, 16] che le applicazioni web siano predisposte alla presenza di vulnerabilità e che queste siano in larga parte indipendenti dalle tecniche di implementazione delle stesse WA; allo stesso modo sappiamo come gran parte degli errori di programmazione siano frutto di una frettolosa e poco attenta progettazione dell'applicazione, che invece dovrebbe essere scritta anzitutto con criteri di correttezza funzionale e sicurezza [17, 18]. Da queste considerazioni, deriva il concetto che lo stesso codice

dell'applicazione fa parte del perimetro di sicurezza del nostro sistema [14], e di conseguenza le WA costituiscono senza dubbio uno dei terreni più fertili per l'exploiting di vulnerabilità, per differenti ragioni:

- *Internet* nasce come rete trusted per lo scambio di informazioni: l'espansione a ritmi esponenziali delle infrastrutture hardware e software ha tuttavia snaturato questa idea: il principio base del TCP, secondo uno dei pionieri di Internet Jon Postel, era (ed è) di fare molta attenzione ai dati inviati, e di accettare di buon grado i dati ricevuti (*TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others* [4]): questo poteva valere agli albori del WWW, ma non certo ora: anzi, si avverte la necessità di ampliare il principio prima enunciato, aggiungendo una fase di validazione intensiva di quello che si è ricevuto dall'esterno;

- in secondo luogo, le WA possono vantare un grandissimo bacino di utenza, essendo (appunto) fruibili via web e intrinsecamente aperte alla comunicazione remota;

- la distanza fisica, propria delle applicazioni remote, è un ulteriore elemento di incoraggiamento alle azioni illecite nel mondo del web, poichè offre l'opportunità di mascherare la propria identità, e sfruttare la molteplicità dei collegamenti nel cammino end-to-end per alterare diversi parametri dei messaggi in transito;

- le WA hanno spesso accesso, per necessità oggettive, a basi di dati che possono contenere dati riservati o interessanti per eventuali aggressori;

- esistono differenti tecnologie per la costruzione di WA, tutte orientate all'immediatezza e alla semplicità d'uso: costruire applicazioni web è spesso veloce ed economico anche se non sempre le persone che scrivono il software sono pienamente consapevoli dei rischi;

- il fatto che la rete interna sia sempre più sicura (si pensi a diverse soluzioni quali *antivirus*, *firewall* e *IDS*) sposta automaticamente l'in-

teresse degli aggressori sulle interfacce pubbliche delle WA, che spesso sono la via più veloce per accedere ai sistemi interni [13, 19];

- a livello commerciale, le aziende hanno iniziato recentemente a sfruttare le potenzialità della Rete, e il desiderio di essere sul web spinge costantemente nuove società a creare finestre sul proprio sistema informativo aperte a tutti (anche a causa della convenienza del mezzo informatico).

Nella definizione di standard di sicurezza, OWASP propone una classifica [14] in cui vengono enumerate le vulnerabilità che affliggono i software su Internet. In questo documento, in costante evoluzione, sono definiti in maniera ordinata le criticità delle applicazioni, fornendo un ottimo punto di partenza per la preparazione di checklist. Se in un primo momento sembra essere l'ennesima pubblicazione di problematiche note da anni, ci si accorge ben presto di quanto sia indispensabile la definizione formale delle vulnerabilità; definire una lista comune di problematiche è infatti il primo passo per discutere ed affrontare il problema a livello di comunità, come avviene per le questioni legate al networking con la lista SANS [20].

Iniziamo quindi la nostra panoramica sulle dieci vulnerabilità più critiche, partendo dal problema più diffuso ed in un certo senso più pericoloso; le successive problematiche verranno trattate per completezza, anche se non in maniera esaustiva.

2.2.1 Unvalidated input

Rientrano sotto questa categoria tutti i problemi legati alla mancanza di validazione dell'input o alla validazione parziale. Come illustrato la comunicazione tra client e server avviene attraverso l'invio di messaggi. Un utente, e quindi anche un potenziale aggressore, può modificare in parte o completamente la richiesta inoltrata verso il server web, plasmando a piacere il contenuto di tali parametri. La possibilità di editare le variabili nell'header HTTP, il valore dei campi ritornati da un form, e così via, apre l'enorme problematica legata alla validazione dei parametri che l'applicazione, sul server potrà accettare. Nel caso in cui il controllo sui valori in ingresso non sia ben

implementato, correremo il rischio di esporre la nostra applicazione ad una serie di attacchi noti che vanno dalla classica SQL injection, al Cross Site Scripting (XSS), Buffer Overflow, Format String e così via [21, 22].

Questo genere di vulnerabilità è il principale campo di interesse di questa tesi poichè risultano compatibili con il rilevamento effettuato tramite strumenti automatici di ricostruzione delle variabili ed eseguito attraverso approssimazioni successive del linguaggio contenuto. Gli strumenti che analizzano staticamente il codice tramite ricostruzione permettono principalmente il rilevamento di vulnerabilità legate al "mal filtraggio" delle stringhe di input. Nei nostri esempi ci rifaremo quindi a casi comuni di *parameter tampering*. In particolare analizzeremo in dettaglio attacchi di tipo *Sql Injection*, *Path Traversal* e infine un caso di *Cross Site Scripting* derivanti da bug su applicazioni reali; proprio le prime due famiglie di vulnerabilità elencate saranno oggetto dell'implementazione di due plugin a corredo dello strumento software realizzato.

SQL Injection

È una tecnica di attacco basata sull'inserimento di valori *untrusted* all'interno di una query SQL, con l'obiettivo di eseguire del codice SQL arbitrario [23]. L'efficacia di questa tecnica, nelle sue numerose varianti [24, 25, 26, 27, 28], deriva dal fatto che le applicazioni web sono, ormai sempre più spesso, affiancate da un database relazionale il quale potrebbe contenere informazioni interessanti per l'attacker; è infatti un tipo di vulnerabilità estremamente diffusa e pericolosa poichè è direttamente legata alla problematica dell'*information disclosure*: attraverso l'exploit di SQL Injection è possibile arrivare a carpire preziose informazioni che dovrebbero rimanere riservate.

Consideriamo il problema presente in una vecchia versione di *Squirrel Mail* [29] e in particolare dell'address book, il quale si appoggia ad un db (MySQL) per il salvataggio delle informazioni. Riportiamo per completezza la struttura della tabella "address" alla quale faremo riferimento.

```
CREATE TABLE address (
owner varchar(50) default NULL,
nickname varchar(50) default NULL,
firstname varchar(50) default NULL,
lastname varchar(50) default NULL,
email varchar(50) default NULL,
label varchar(50) default NULL )
```

Nella versione 1.15.2.1 all'interno della pagina *abook_database.php* è presente la seguente query:

```
$query = sprintf("SELECT * FROM %s WHERE owner='%s'
AND nickname='%s'", $this->table, $this->owner, $alias);
$res =$this->dbh->query($query);
```

In nessun punto precedente del codice viene effettuato un controllo sul contenuto delle variabili e poichè tale valore è modificabile dall'utente, il mancato controllo di validazione dell'input rappresenta la giusta collocazione per un attacco di questo tipo. Se per esempio la variabile $alias contenesse la seguente stringa:

```
' UNION ALL SELECT * FROM address WHERE ''='
```

Tale segmento di codice genererebbe la seguente istruzione SQL:

```
SELECT * FROM address WHERE owner='me'
AND nickname='' UNION ALL SELECT * FROM address WHERE ''=''
```

Supponendo che non ci sia nessun utente con il campo nickname vuoto, la prima SELECT non restituirà risultati mentre la seconda, poichè costruita con una clausola WHERE incondizionata, ritornerà tutte le tuple contenute nel database.

L'utilizzo del comando SQL di aliasing (AS) permette poi di bypassare i problemi di visualizzazione dei risultati della query. Una semplice risoluzione di tale problema (presente nella versione successiva, la 1.15.2.2) utilizza un'istruzione di verifica *quoteString*, presente nella libreria PEAR MDB [30],

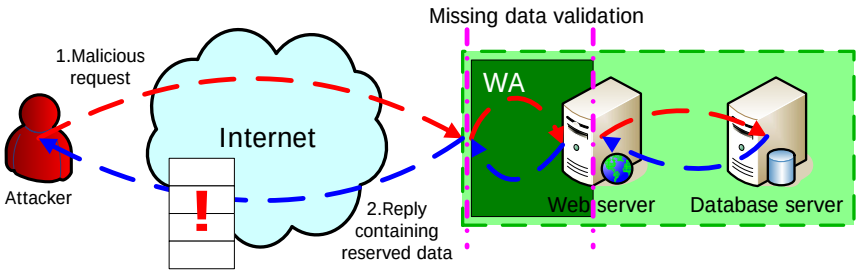

Figura 2.3: Schema di un attacco di tipo SQL Injection

che effettua l'escape del carattere apice. Una soluzione semplice ad un problema concettualmente semplice ma spesso difficilmente individuabile, senza l'ausilio di strumenti automatici di controllo, in progetti di una certa entità.

Concludiamo la presentazione di questa tipologia di vulnerabilità con un esempio che, a dispetto di eventuali limitazioni sul numero massimo di caratteri forgiabili, permette la creazione di un comando SQL breve e al tempo stesso estremamente dannosa per la WA:

```
'; shutdown --
```

In questo caso la query porta all'esecuzione del comando di *shutdown* dimostrando come sia possibile utilizzare un'istruzione SQL per effettuare un attacco di tipo DoS (Denial of Service); con questa semplice stringa utilizzata come parametro di ingresso sarebbe possibile sconnettere dalla rete un servizio web con ingenti perdite economiche e di immagine.

Path Traversal

Consideriamo una nuova applicazione e vediamo, ancora una volta come, semplici dimenticanze sulla validazione degli input, producono problemi seri per quanto riguarda la sicurezza dell'applicazione. In questo caso parliamo di *phpMyAdmin*, un tool sviluppato in PHP per la completa amministrazione di *MySQL* attraverso un'interfaccia web. Sino alla versione 2.5.x di tale software è presente un problema di sicurezza rientrante nella categoria di vulnerabilità che va sotto il nome di Path Traversal, tramite il quale risulta possibile effettuare il listing di una directory del web server che ospita

l'applicazione. Inviando un richiesta al file *db_details_importdocsql.php*, con i seguenti parametri:

```
db_details_importdocsql.php?do=import&docpath=PATH
```

e sostituendo a *PATH* l'opportuno path per la directory di cui vogliamo avere il listing, riusciamo senza nessun problema ad ottenere il risultato sperato. Sostituendo, per esempio "*../../../*" è quindi possibile mostrare il contenuto della root. Sebbene in questo modo sia unicamente possibile visualizzare l'elenco dei file presenti, e non richiedere i file stessi, tale attacco può comunque essere un'ottima base di partenza per ottenere maggiori informazioni sulla vittima nonchè per scoprire eventuali pagine pubblicate nel server web (magari usate come test) ma non collegate pubblicamente al sito.

Cross Site Scripting (XSS)

All'interno di questa tipologia di attacchi rientrano tutti quei casi in cui l'aggressore ha la possibilità di inserire, all'interno della WA, del codice arbitrario HTML e/o JavaScript così da modificarne il comportamento [31, 32] per perseguire i propri fini illeciti. A titolo di esempio, mostriamo un'altra vulnerabilità presente in una vecchia versione di *Squirrel Mail* [29] in cui c'è un problema di XSS, dovuto ad un controllo non adeguato sui parametri in ingresso.

```
$day=$_GET['day'];
$month=$_GET['month'];
$year=$_GET['year'];
echo"<a href=\"day.php?year=$year&amp;"
echo"month=$month&amp;day=$day\">";
```

In questa parte dello script, le variabili rappresentanti il giorno, mese e anno vengono prelevate da una precedente richiesta, per comporre una nuova pagina HTML. Senza aver effettuato alcun controllo sul contenuto delle variabili queste potranno essere usate per inserire del codice malevolo da far eseguire direttamente all'utente stesso, sul proprio client. Inserendo quindi una stringa del tipo:

```
http://.../event_delete.php?year=><script>code();</script>
```

e convincendo l'utente ad aprire tale URL (per esempio tramite tecniche di social engineering) è possibile eseguire comandi che hanno gli stessi privilegi del server web stesso. Questa situazione è l'ideale per attacchi di *session hijacking* in cui poter prelevare cookie ed altre informazioni sensibili all'utente stesso. L'HTML di risposta alla richiesta HTTP dell'utente è infatti

```
<a href="day.php?year=><script>code();</script>
```

e il parametro year, ora riempito con la funzione *code* è il vettore attraverso il quale eseguire del codice arbitrario.

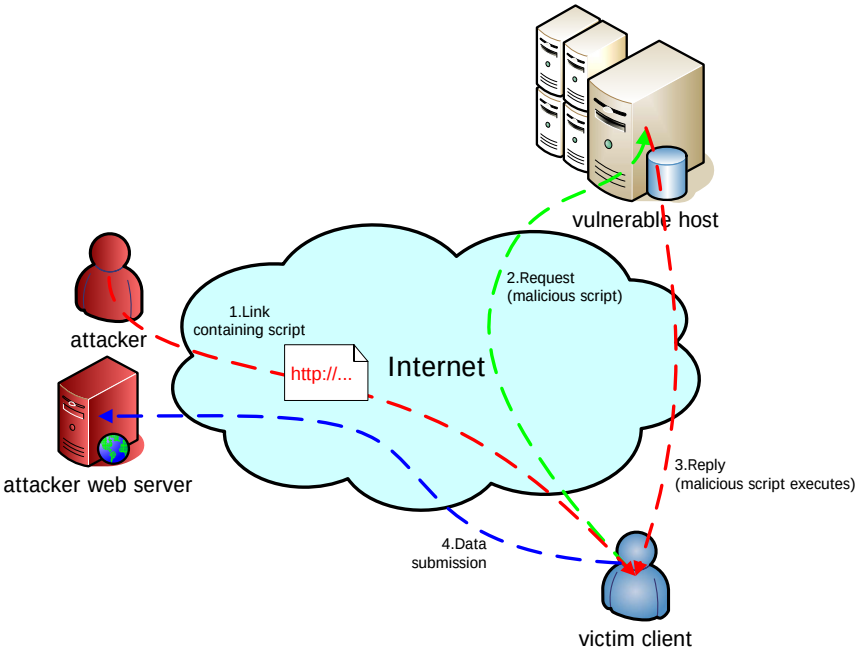

Figura 2.4: Schema di un attacco di tipo XSS

Va anche notato come grazie ad alcune astrazioni HTML sia possibile variare la formattazione del link, facendo richiamare uno script salvato altrove o mimetizzando la sua chiamata, con costrutti quali:

```
<img src="javascript:...">
<script src="http://...">
```

Analizzando ora le nuove versioni dell'applicativo, ci si rende conto di quanto questo problema sia facilmente risolvibile, attraverso l'utilizzo di parametri unicamente numerici ed una serie di opportune funzioni di validazione (es: *is_numeric($_GET['month'])*).

2.2.2 Altre vulnerabilità dalla OWASP TopTen

Come anticipato, concluderemo la presentazione delle vulnerabilità presenti nella OWASP Top Ten per completezza della trattazione pur tuttavia dedicando solamente un piccolo accenno alle rimanenti categorie.

2. **Broken Access Control:** le politiche di accesso ai dati riservati non sono implementate con rigore, e un attacker può entrare in possesso di dati "privati", o impadronirsi di account di altri utenti. Spesso è un problema sottovalutato, poichè lo stadio di access control è collocato al di sopra di quello di autenticazione. Ma una volta autenticato un utente, è necessario avere un modello che stabilisca in modo rigoroso, coerente e unitario, privilegi e permessi dell'utente stesso. Spesso, infatti, le regole che stabiliscono i privilegi sono frammentate e contraddittorie, e certi diritti possono essere accordati a utenti che non dovrebbero usufruirne.

3. **Broken Authentication And Session Management:** i token di sessione e i dati relativi alle identità non sono conservati e amministrati correttamente; è possibile il furto di identità sia attraverso username e password, sia attraverso chiavi di sessione.

4. **Cross Site Scripting (XSS):** come già accennato parlando di problemi di validazione dell'input, questa vulnerabilità viene classificata in maniera separata per la particolarità del meccanismo di attacco. Ai fini della nostra analisi può essere semplicemente considerata come una qualsiasi altra problematica di errata validazione dell'input.

5. **Buffer Overflows:** è un attacco che sfrutta una validazione errata dell'input, in cui non si controlla la lunghezza della sequenza di dati ricevuti dall'applicazione. Un utente, per errore o maliziosamente, può

modificare la parte di codice in esecuzione nello stack andando a cambiare il flusso di esecuzione del software. Tecnicamente esistono diverse strategie - *stack overflow*, *heap overflow*, *format string* - che possono però essere ricondotte ad un'unica grande famiglia di vulnerabilità in cui l'aggressore può riuscire ad eseguire comandi arbitrari. Nel caso delle applicazioni online è possibile riscontrare tale vulnerabilità sia nel web server stesso che all'interno di librerie custom utilizzate dalla web application. È opportuno ricordare che, con l'avvento delle tecnologie Java (JSP) e gli altri linguaggi interpretati, tale problematica sta lentamente diminuendo e diventando sempre meno importante.

6. **Injection Flaws:** si tratta di un caso particolare di tampering dei parametri e mancata validazione; in questo caso la WA riceve dall'utente un input all'interno del quale è stato inserito un comando eseguibile da qualche componente normalmente contattato dalla stessa WA. In questo modo, con i privilegi della WA, verrà eseguito un comando - potenzialmente dannoso - deciso dall'utente finale e non previsto dal codice dell'applicazione [33].

7. **Improper Error Handling:** gli stati di errore non sono gestiti correttamente; l'attacker può, in questo caso, ottenere informazioni sul sistema partendo dai messaggi di errore. Arrivare a rubare dati dal server o eseguire codice su di esso implica una piena conoscenza del sistema che bisogna aggredire e dei meccanismi di sicurezza da aggirare; all'aggressore servono informazioni sui servizi presenti, sulle loro versioni e configurazioni oltre che sulla struttura delle directory. Una gestione non corretta degli errori può rivelare in maniera diretta tali informazioni, trasformando un errore apparentemente innocuo in un messaggio utilissimo per scopi illeciti.

8. **Insecure Storage:** la WA affida la protezione dei dati sensibili ad algoritmi di cifratura o politiche di accesso mal progettate, risultando in una generale mancanza di protezione dei dati stessi.

9. **Denial of Service (DoS):** Come tutti i servizi di rete, anche le applicazioni web, soffrono degli attacchi di tipo DoS in cui, attraverso

l'esaurimento delle risorse del sistema, si satura la sua capacità fino a determinarne il blocco temporaneo. In questa situazione i sistemi diventano inutilizzabili per gli utenti legittimamente attivi. Una grossa quantità di richieste HTTP è un primo e semplice esempio di come poter saturare le risorse di un server web, bloccando l'accesso per tutti gli altri utenti.

10. **Insecure Configuration Management:** i server che ospitano le WA possono avere molte opzioni di configurazione che minano la stabilità e la sicurezza della WA se usate a proprio vantaggio da parte di un eventuale aggressore.

2.3 Rischi

Tutte le problematiche precedentemente illustrate dimostrano le innumerevoli modalità di attacco sfruttabili da un aggressore e nel contempo servono a comprendere l'importanza di metodologie preventive e proattive per combattere il fenomeno. La sicurezza informatica ha, come obiettivo principale, quello di garantire, riducendo i rischi, un adeguato grado di protezione dei beni, mediante l'attuazione di un progetto di sicurezza globale; tale pianificazione, partendo dalla definizione di una politica di sicurezza, deve tener conto di tutti gli aspetti del problema e pervenire ad un livello di protezione, organizzativo ed informatico, che possa essere monitorato nel tempo.

Per meglio comprendere la relazione che intercorre tra rischio e vulnerabilità risulta necessario definire con precisione queste espressioni, oltre ad introdurre una nuova locuzione. In termini di sicurezza informatica definiamo il **rischio** come l'esposizione di un'organizzazione a perdite o danni. Con il vocabolo **minaccia** intendiamo invece quei fattori esterni che rappresentano un pericolo per la nostra organizzazione. Una **vulnerabilità** è quindi una debolezza nel sistema che può essere utilizzata (*exploit*) da una minaccia (*aggressore*). L'atto con cui una minaccia sfrutta una vulnerabilità per trasformare il rischio in danno si chiama *attacco* o *disastro*.

Nel mondo dell'information technology definiamo la sicurezza il raggiungimento di tre obiettivi prioritari:

Confidenzialità/Riservatezza (Confidentiality): solo le persone autorizzate possono accedere al sistema informativo o alle sue risorse

Integrità (Integrity): solo le persone autorizzate possono modificare le componenti del sistema e le sue risorse, solo nelle modalità per cui sono state autorizzate a procedere

Disponibilità (Availability): il sistema deve fornire i servizi richiesti in un tempo "ragionevole" secondo gli specifici requisiti

Per assicurare queste caratteristiche ad ogni sistema informativo devono essere adottate misure tecnologiche e organizzative atte a scongiurare il rischio di attacco informatico. Tra le soluzioni preventive risulta utile citare le attività di auditing sul codice sorgente, svolte durante e successivamente lo sviluppo del software, che dovrebbero prevenire ed arginare eventuali problemi ancora prima della messa in opera del servizio. Questo lavoro di revisione, in contesti professionali, è un processo per lo più svolto manualmente utilizzando strumenti automatici a supporto del tester.

Il nostro lavoro di tesi si inserisce in questo contesto, cercando di fornire uno strumento utile agli esperti di "secure coding" al fine di identificare quante più vulnerabilità e bug presenti nel codice possibile.

2.3.1 Diffusione del fenomeno

Prima di analizzare in dettaglio le differenti modalità di testing sulle WA e le relative limitazioni, risulta interessante osservare alcuni dati per capire la reale diffusione del problema e le strategie per combatterlo. Secondo il capitolo italiano di OWASP [34] che ha realizzato uno studio su 47 WA utilizzate normalmente da alcune realtà industriali italiane risulta chiaro come le problematiche di input validation e parameter tampering, ovvero quelle che scaturiscono dall'uso di parametri che ricevono assegnamenti da input esterno e che non subiscono un'adeguata trafila di controllo, siano ancora [35] i problemi maggiori. Come si può osservare dal diagramma 2.5 anche l'errata gestione dei messaggi di errore è una problematica piuttosto sentita sebbene di per sè non rappresenti un reale rischio per le WA stesse.

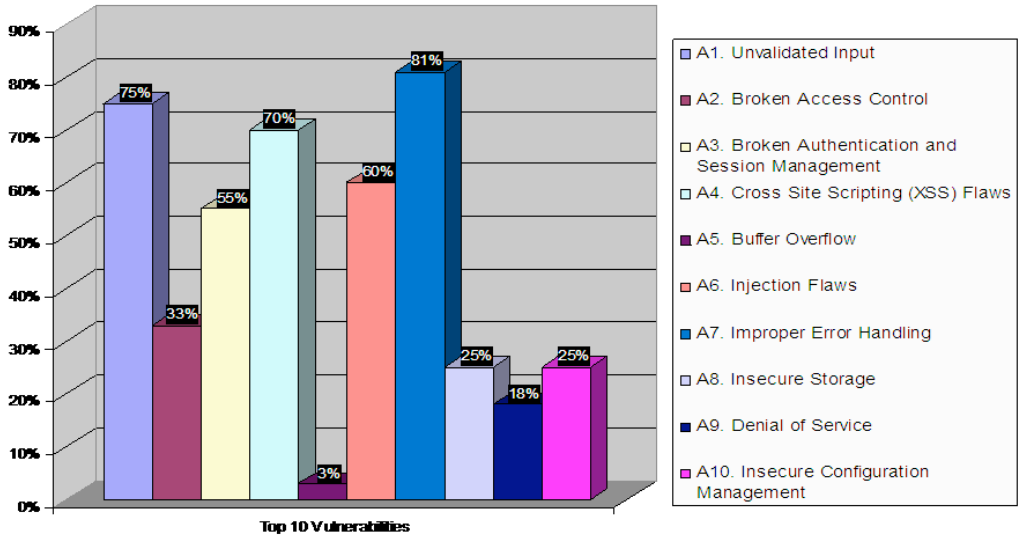

Figura 2.5: Vulnerabilità critiche nel panorama delle WA in Italia

Basandosi su di un ulteriore studio svolto da WhiteHat Security [36] (società operante nel settore della sicurezza informatica) su di un campione di WA che sono state analizzate dai loro esperti, si osservano dei risultati interessanti per quanto riguarda il rapporto tra testing manuale e utilizzo di strumenti di auditing automatici. Nel 36% delle WA analizzate gli esperti di WhiteHat Security non hanno identificato vulnerabilità che invece sono state rilevate dagli strumenti automatici. Nel 17% delle WA il personale tecnico ha invece riscontrato tutte le problematiche mentre, per gli scanner, il software risultava esente da errori. Infine nel 47% dei casi il lavoro svolto da tecnici e l'utilizzo di strumenti di revisione è stato complementare, riuscendo così a risolvere gran parte dei problemi di sicurezza presenti nel software analizzato.

Capitolo 3

Verifica del Codice

Negli ultimi anni l'industria della sicurezza sta cambiando approccio rispetto alla percezione e alla risoluzione dei problemi di sicurezza nelle infrastrutture informatiche. Sino a qualche anno fa, le risorse dei dipartimenti preposti alla messa in sicurezza degli impianti venivano utilizzate per quello che possiamo definire con il termine "network security"; oggi invece la situazione è molto diversa e quello che viene richiesto è spesso di garantire sicurezza a livello applicativo. Questa richiesta del mercato ha generato un interesse sempre maggiore verso aziende, università e comunità della Rete che si occupano di "software security"; in maniera proporzionale sono apparse nuove tecnologie, nuove metodologie di test [37] e nuovi strumenti per fronteggiare questa necessità: primi tra tutti gli strumenti di revisione automatica del codice.

Per gli applicativi sviluppati attraverso linguaggi con organizzazione della memoria a pila, una delle vulnerabilità più facile da sfruttare è il *buffer overflow*; tuttavia per questo genere di problematiche esistono già teorie e tool che cercano di dare una risposta a questo problema [38], soprattutto per quanto riguarda l'analisi del codice C e C++ [39, 40, 41]. Il problema della ricerca di vulnerabilità nelle WA è invece un problema recente, che trova prevalentemente soluzione in ambito commerciale, anche se esistono tool *Open Source* [42] e progetti di ricerca in ambito universitario. Per molti degli strumenti che considereremo in realtà sarebbe meglio parlare di ten-

tativi di soluzione, data la complessità e la forte mutabilità del problema: spesso ci si trova davanti ad applicazioni che si limitano a cercare casi noti di vulnerabilità, mettendo quindi in campo una vasta base di *conoscenza* piuttosto che di *intelligenza*. Questa è anche una conseguenza dell'approccio utilizzato nella maggioranza dei casi, che è di tipo *online*: sfruttando la connessione al server e interpretando le risposte di certi messaggi formattati accuratamente, è molto facile ottenere informazioni che sono al tempo stesso affidabili, ma spesso troppo *generiche* [43] e prive di reale utilità durante la fase di test.

La scelta sull'utilizzo di strumenti automatici o semiautomatici è condizionata da un numero molto alto di fattori: prima di tutto è necessario identificare le tecnologie utilizzate e scegliere i tool più adatti al contesto applicativo; spesso però ci sono anche aspetti che si allontanano dalle considerazioni puramente tecniche come la disponibilità o meno del codice sorgente, l'assoluta necessità di testare il software sull'ambiente finale di deploy, di effettuare la revisione in diverse fasi del processo produttivo [44] dell'applicativo stesso e così via. Le considerazioni appena fatte sono uno dei principali motivi per cui spesso non risulta possibile utilizzare un singolo prodotto per risolvere tutti i problemi di una WA; inoltre è giusto ricordare che spesso alcune problematiche legate ad errori di design non sono comunque identificabili in maniera automatica. È facile intuire che *non esiste una pallottola d'argento per la sicurezza in ambito web, anche se alcune società del settore tendono a farlo credere. Per risolvere questa sfida occorrono persone capaci, grande conoscenza e ottima formazione, ottimi processi e il meglio della tecnologia* [**Mark Curphey**, fondatore di OWASP].

Parlando di strumenti per l'analisi e la scansione di WA è inevitabile dover affrontare problemi di nomenclatura e categorizzazione, poichè esistono numerosi tool che utilizzano metodologie di analisi differenti per raggiungere il medesimo scopo: il rilevamento di vulnerabilità. In questa sede utilizzeremo una divisione che risulta accettata da gran parte degli esperti del settore, sebbene non sia stata ancora formalizzata da nessuna organizzazione; per conformità utilizzeremo la terminologia inglese per identificare le varie categorie. Questa divisione raggruppa gli strumenti di analisi in set-

te gruppi: *source code analyzer*, *web application scanner*, *database scanner*, *binary analysis tool*, *runtime analysis tool*, *configuration management tool*, *HTTP proxy*. Per meglio evidenziare le caratteristiche del metodo e dello strumento software che abbiamo sviluppato, illustreremo solamente alcune di queste categorie segnalando l'area di azione del nostro tool e le differenze rispetto ad altri analizzatori.

3.1 Source code analyzer

Con il termine "analizzatori di codice sorgente" si intendono tutti gli strumenti che utilizzano uno dei seguenti procedimenti di analisi:

- Revisione statica del codice sorgente
- Revisione dinamica del codice sorgente

Gli analizzatori statici si basano principalmente sulla ricerca di pattern all'interno del codice sorgente al fine di identificare modelli di programmazione errati che possono introdurre problematiche di sicurezza. I più moderni strumenti di revisione appartenenti a questa famiglia (come il nostro tool) utilizzano dei processi di analisi che permettono di tracciare il flusso dei dati attraverso il codice in maniera da fornire un'analisi più accurata e completa.

I primi strumenti di analisi statica sono sul mercato ormai da molto tempo, sebbene non siano realmente utilizzabili in contesti applicativi dove bisogna analizzare applicazioni di grosse dimensioni: essi si limitano alla ricerca di pattern (nei casi più semplici, singole stringhe) provvedendo a segnalare al tester ogni singola occorrenza. Se per esempio consideriamo la pericolosa funzione *strcpy*, con uno strumento di questo tipo saranno semplicemente riportati tutti i punti all'interno del codice sorgente in cui viene effettivamente usata la funzione *strcpy*, lasciando al revisore l'ingrato compito di differenziare i falsi positivi dalle vere e proprie vulnerabilità. Molti degli strumenti avanzati, come detto, cercano di ridurre significativamente il numero di falsi positivi attraverso *dataflow analysis*; con questo termine si indica una particolare tecnica di indagine che utilizza le teorie legate al *control flow graph* per determinare il valore di alcuni parametri del programma.

A differenza dei precedenti, gli analizzatori dinamici attuano un'analisi in profondità nel codice sorgente cercando di ricostruire lo stack delle chiamate a runtime, determinando se la specifica invocazione viene effettivamente raggiunta dai dati ricevuti in ingresso. I più moderni strumenti si integrano perfettamente con i debugger [45, 46] e permettono di tracciare e segnalare gli errori nel momento in cui effettivamente si verificano. Il principale problema di questa classe di strumenti è legato alla lentezza di scansione e alla necessità di fornire un dataset significativo di valori in input.

In generale i "source code analyzer" statici e dinamici risultano strumenti estremamente interessanti in quanto possono essere utilizzati sin dai primi momenti dello sviluppo del software. Inoltre la segnalazione di errori a questo livello permette di ridurre notevolmente i costi del processo produttivo, il che spesso fa preferire questa tipologia di analizzatori ad altre. Di contro, nel caso di grandi progetti, spesso non è possibile compilare completamente l'intera applicazione per effettuare il testing nelle varie fasi del processo produttivo, il che limita l'applicabilità del metodo. C'è poi da ricordare che spesso questi tool segnalano generalmente errori con granularità pari ad una singola linea di codice sorgente, mentre difficilmente riescono ad evidenziare problematiche che interessano successive istruzioni appartenenti a differenti funzioni o file.

3.2 Web application scanner

Gli scanner per WA, conosciuti anche come *black-box* scanner simulano la normale interazione tra web browser e server remoto cercando componenti con vulnerabilità note oppure effettuando l'invio di payload ritenuti potenzialmente pericolosi. L'efficacia di questi strumenti è ovviamente legata alla base di conoscenza: più questa risulta aggiornata e personalizzata per la particolare WA, migliori saranno i risultati. Generalmente questi software sono molto semplici da utilizzare poichè non richiedono nessuna interazione da parte dell'utente. L'attendibilità dei report generati dipende notevolmente dal genere di applicazione che si sta testando: mentre i siti web statici sono facilmente analizzabili dagli spider usati all'interno di questi tool, non si può dire lo stesso per le WA che fanno un uso intensivo di JavaScript,

AJAX, Adobe Flash, form di autenticazione e percorsi di navigazione che richiedono una costante interazione con l'utente. Infine, un grande difetto legato alle scansioni di tipo black-box è l'impossibilità di determinare con precisione dove effettivamente risiedano gli errori evidenziati, lasciando agli sviluppatori il compito di scovare tali problematiche all'interno del codice sorgente.

3.3 Altri strumenti di analisi

Oltre alle categorie che abbiamo già segnalato, vogliamo brevemente accennare alle altre tipologie di strumenti utilizzabili da tester e sviluppatori.

Con il termine *binary analysis tool* si definiscono tutti quegli strumenti che cercano di scovare falle di sicurezza all'interno di programmi binari; il campo di applicazione di questi strumenti è generalmente limitato alle applicazioni C e C++. Questi software cercano di determinare le interfacce pubbliche delle applicazioni sotto analisi al fine di inviare dei valori di input notoriamente pericolosi, in attesa che l'applicazione stessa dia segnali di mal funzionamento oppure si blocchi completamente.

Gli strumenti denominati *runtime analysis tool* invece possono essere facilmente confusi con quelli che analizzano in maniera dinamica il codice sorgente; in questo caso però i runtime analysis tool non cercano di determinare le vulnerabilità, limitandosi a evidenziare informazioni e casi di test che potrebbero generare a runtime degli errori.

Un'altra classe è quella che va sotto il nome di *database scanner* i quali replicano l'interazione tra client e server SQL, in maniera da poter inviare delle query che possono testare la configurazione dei dbms (procedure, utenti, ruoli, privilegi) oltre ad effettuare l'invio di classiche stringhe riconducibili ad attacchi SQL Injection.

I *configuration analysis tool* operano effettuando dei test sulle configurazione delle WA ma spesso anche sui server web stessi; i classici problemi rilevati da questo genere di tool sono errori di configurazioni o impostazioni di default pericolose. L'ultima categoria che analizziamo va sotto il nome di *proxy*, i quali si occupano di intercettare il traffico tra web browser e server web, permettendo al tester di modificare parzialmente le richieste HTTP;

questo genere di applicazioni sono utilissime e molto versatili ma implicano una conoscenza molto approfondita sull'argomento in quanto il lavoro deve essere svolto in gran parte manualmente.

3.4 Soluzioni esistenti

Dopo aver enumerato le differenti classi di scanner e analizzatori, vogliamo fornire una panoramica degli strumenti software esistenti dividendoli tra strumenti commerciali, Open Source o progetti di ricerca; per mantenere la continuità logica del discorso indicheremo opportunamente la categoria di appartenenza.

3.4.1 Applicativi commerciali

- **Secure Software CodeAssure** (source code analyzer)
  www.securesoftware.com/products

- **Ounce Labs Prexis** (source code analyzer)
  www.ouncelabs.com/prexis_engine.html

- **Fortify Software Source Code Analysis** (source code analyzer)
  www.fortifysoftware.com/products/sca.jsp

- **Coverity Prevent/Extend** (source code analyzer)
  www.coverity.com/products/index.html

- **Compuware DP SecurityChecker** (source code analyzer)
  www.compuware.com/products/devpartner/securitychecker.htm

- **SPI Dynamics WebInspect** (wa scanner)
  www.spidynamics.com
  Tool molto potente, controlla circa 1500 vulnerabilità note su web server e applicazioni, e permette anche la ricerca di casi triviali di vulnerabilità di passaggio dei parametri, hidden field, password guessing. Permette controlli customizzabili, ma estremamente semplici. La scansione, come in tutti gli strumenti appartenenti a questa categoria, viene effettuata inviando per ogni possibile parametro in ingresso una

serie di payload potenzialmente pericolosi; al contrario, il nostro tool, cercherà di individuare puntualmente i metodi insicuri.

- **N-Stealth Security Scanner** (wa scanner)
  www.nstalker.com
  Prodotto principalmente destinato all'analisi dei web server; dichiara di effettuare ricerche contro oltre 30000 casistiche di problemi su HTTP e HTTPS, oltre a permettere la scrittura di firme di vulnerabilità "personalizzate'. Attraverso un motore di aggiornamento compatibile con la notazione CVE (Common Vulnerabilities and Exposures) è possibile mantenersi al passo con le vulnerabilità scoperte. Permette inoltre dei security test veloci che vanno a verificare il web server secondo le vulnerabilità presenti nella nota Top20 SANS/FBI [20].

- **NGSSoftware Typhon** (wa scanner)
  www.ngssoftware.com
  Evoluzione di Cerberus Internet Scanner (CIS), punta sulla qualità dei controlli più che sul loro numero. Comprende anche dei controlli a livello di applicazioni web, sebbene effettui test anche a livello di networking. Permette la personalizzazione del formato dei report.

- **Watchfire AppScan** (wa scanner)
  www.watchfire.com
  Ricerca comuni vulnerabilità che affliggono i web server ed a livello applicativo simula situazioni di attacco alla ricerca di falle di sicurezza. Permette l'analisi delle dieci vulnerabilità critiche individuate da OWASP oltre a numerose altre; interessante il supporto legato alle nuove tecnologie del web (XML/SOAP Test, XPath Injection) e la buona capacità di riconoscimento di XSS. Un eccellente report oltre ad una buona velocità nel caso di piccole applicazioni rendono questo prodotto abbastanza interessante.

- **Acunetix Web Vulnerability Scanner** (wa scanner)
  www.acunetix.com
  La capacità di riconoscimento dichiarata dal produttore spazia dal Cross Site Scripting, SQL Injection, Code execution, File Inclusion al-

l'interessante "Google hacking". Con questo termine si identifica una tecnica tramite la quale eventuali aggressori possono trarre informazioni critiche utilizzando semplicemente un motore di ricerca. Questo prodotto indicizza i contenuti dell'applicazione analizzata tramite un crawler e poi esegue le query classiche utilizzate dagli aggressori. Interessante anche la funzione HTTP Fuzzer con cui è possibile creare delle regole personalizzate che generino degli attacchi dinamici.

- **Application Security Inc. AppDetective** (database scanner)
  www.appsecinc.com/products/appdetective/index.shtml

- **DBAppSecurity MatriXay** (database scanner)
  www.dbappsecurity.com/index.html

- **BugScan with IDAPro** (binary analysis)
  www.logiclibrary.xom

- **Compuware BoundsChecker** (runtime analysis)
  www.compuware.com/products/devpartner/studio.htm

3.4.2 Applicativi Open Source

- ***R*ough *A*uditing *T*ool for *S*ecurity** (source code analyzer)
  www.securesoftware.com/resources/download_rats.html
  È uno strumento appositamente studiato per la scansione di sorgenti C, C++, Perl, PHP e Python. È in grado di segnalare numerosi errori di mal programmazione, buffer overflow oltre a problematiche legate a *race condition*. A differenza del nostro strumento, RATS si concentra su linguaggi differenti da Java e su tipologie di problematiche che stanno diventando sempre meno frequenti in ambito web.

- **FlawFinder** (source code analyzer)
  www.dwheeler.com/flawfinder

- **FindBugs** (source code analyzer)
  findbugs.sourceforge.net

- **Nikto** (wa scanner)
  www.cirt.net/code/nikto.shtml
  Ricerca errori di configurazione, file e script noti, software obsoleto, su HTTP e HTTPS; effettua basilari operazioni di scansione delle porte, ed è aggiornabile automaticamente via Internet. Per le ricerche utilizza un componente, *libwhisker*, direttamente mutuato da un altro progetto [47].

- **BurpProxy** (wa scanner, proxy)
  portswigger.net
  Permette di intercettare le richieste HTTP/HTTPS che dal browser vengono inviate verso il server; in questo modo è possibile ispezionare e modificare tutti i parametri. Dispone inoltre di un comodo spider integrato e di uno strumento per effettuare attacchi in maniera automatizzata. Con quest'ultimo componente della suite (Burp Intruder) è possibile selezionare dei parametri in maniera dinamica, modificarne il contenuto e inoltrare più volte la richiesta verso il server; una comoda funzione di riconoscimento basata su espressioni regolari determina il risultato della pagina e fornisce un comodo report.

- **OWASP Pantera** (wa scanner, proxy)
  www.owasp.org/index.php/OWASP_Pantera_Web_Assessment_Studio

- **OWASP WebScarab** (wa scanner, proxy)
  www.owasp.org/index.php/OWASP_WebScarab_Project
  WebScarab è un framework per l'analisi di WA scritto interamente in Java e per questo portabile su qualsiasi piattaforma. Permette di intercettare ed analizzare il flusso di informazioni tra client e server; permette di effettuare test manuali ma anche di automatizzare alcune richieste.

- **MetaCoretex** (database scanner)
  www.securityforest.com/wiki/index.php/Category:Enumeration

- **BugScam** (binary analysis)
  www.sourceforge.net/projects/bugscam

- **FoundStone .NETMon** (runtime analysis)
  www.foundstone.com/resources/proddesc/dotnetmon.htm

- **NProf** (runtime analysis)
  www.mertner.com/confluence/display/NProf/Home

- **FoundStone SSLDigger** (configuration analysis)
  www.foundstone.com/resources/proddesc/ssldigger.htm

- **Paros** (proxy)
  www.parosproxy.org/index.shtml

- **Suru Web Proxy** (proxy)
  www.sensepost.com/research/suru

3.4.3 Progetti di ricerca

- **WebSSARI** (source code analyzer) [48]
  Questo interessante tool, a differenza di molte delle soluzioni prima citate e in accordo con la strada che cercheremo di seguire in questo lavoro di tesi, basa il suo funzionamento sull'analisi statica. Ma anzichè ispezionare il controllo del flusso delle istruzioni, analizza il *flusso delle informazioni*. Significa che vengono attribuite classi di sicurezza ai dati (nel caso più semplice solo due, *sicuro* e *insicuro*), e quando operazioni particolari vengono effettuate con dati non ancora ritenuti sicuri viene sollevata una selezione di vulnerabilità.

- **Pixy** (source code analyzer) [49]
  Interessante strumento per la scansione di applicativi PHP al fine di determinare, in maniera statica, la presenza di vulnerabilità di Cross Site Scripting. I risultati teorici [50], implementati poi nel tool, si rifanno alla dataflow analysis. Rispetto al nostro tool, Pixy risulta specifico per una particolare tipologia di vulnerabilità; sebbene il metodo sia generalizzabile, gli studi ed il tool rilasciato si riferisco unicamente alle problematiche di Cross Site Scripting, limitando lo spettro di applicabilità dello strumento.

- **Lapse** (source code analyzer) [51]
  LAPSE è l'acronimo di *L*ightweight *A*nalysis for *P*rogram *S*ecurity in *E*clipse. È stato sviluppato con l'obiettivo di creare uno strumento versatile per l'auditing di applicazioni Java J2EE integrato direttamente in un'importante ambiente di sviluppo come Eclipse[52]; per questa ragione è uno dei progetti che più si avvicina, negli intenti, al nostro strumento, sebbene utilizzi teoricamente altre modalità di analisi che differiscono da quella presentata in questo lavoro di tesi. In particolare LAPSE si focalizza sulla ricerca dei punti di ingresso delle variabili (source), dei metodi potenzialmente pericolosi (sink), cercando di determinare se esistono dei percorsi validi (path source-sink) tra source e sink.

3.5 Metodologia proposta

Dopo aver illustrato lo spazio del problema e le soluzioni software che sono state proposte, possiamo esplicitare gli obiettivi del nostro progetto, del nostro strumento software e la teoria legata alla metodologia proposta.

Il punto da cui la nostra indagine muove il suo primo passo è la necessità di uno strumento che affianchi il programmatore o progettista durante la creazione di una WA, e che sia di supporto al riconoscimento e alla correzione delle falle di sicurezza lasciate aperte inconsapevolmente. Per quanto meticoloso possa essere il lavoro di chi crea un'applicazione, abbiamo visto come una svista o la mancanza di competenze specifiche possa minare la sicurezza dell'intera infrastruttura informatica.

Evidentemente non è possibile scovare alcune particolari vulnerabilità [53] con la sola ispezione del codice sorgente dell'applicazione, poichè talvolta le problematiche dipendono da scelte infrastrutturali che vanno ad intersecarsi tra loro (per esempio la scelta di determinati sistemi operativi, dbms, particolari gestioni dei privilegi d'accesso, etc). Per questa ragione il nostro strumento si occuperà dei cosidetti difetti *tecnici* ma non di quelli *logici*.

3.5.1 Blacklist, whitelist

Nel corso della nostra analisi faremo spesso riferimento ai punti di validazione (*checkpoint*) inseriti dal programmatore per filtrare le variabili in ingresso (parametri dei form, cookie e così via.). A seconda che i checkpoint [54] siano incorporati nell'applicazione, siano solo parzialmente al suo interno, o risultino inseriti tramite chiamate a librerie è possibile distinguere diverse politiche di controllo:

- validazione estesa (in ogni punto di utilizzo dei dati);
- pattern di validazione nel codice;
- componenti dedicati (librerie o J2EE filters);
- plugin del server web;
- application-level firewall.

Nel nostro progetto dovremo considerare queste diverse modalità di filtraggio poichè saranno sorgenti di approssimazione e di imprecisione.Un punto di controllo deve fornire un servizio di verifica di un dato in transito, comportandosi come un *filtro*. I criteri di valutazione utilizzati dai filtri sono catalogabili in:

blacklist: la validazione è effettuata attraverso una lista di casi noti e non omologati alla politica di sicurezza della WA: se un parametro in input corrisponde a uno degli elementi di questa lista, viene bloccato dal checkpoint. I limiti di questa politica sono evidenti: è assolutamente difficile, se non impossibile, mantenere una blacklist aggiornata e completa, poichè spesso non solo devono essere aggiunti nuovi elementi (la scoperta di un nuovo tipo di *bad input*), ma talvolta questi stessi elementi non sono prevedibili a priori. Non si può pretendere di stabilire, a priori, tutto ciò che deve essere considerato sbagliato: è ormai diffusa la consapevolezza che questa tattica sia inadeguata per le dimensioni del problema che bisogna fronteggiare [18];

whitelist: questo approccio è duale rispetto al precedente. Non si valida un dato confrontandolo con una serie di casi "negativi", ma ponendolo

a confronto con l'insieme di tutto ciò che è riconosciuto essere valido. Un errore nell'implementazione della whitelist si riduce in problemi a livello funzionale e non a vulnerabilità software come nel caso delle blacklist. Una volta deciso cosa risulta sicuro per l'applicazione, i nuovi tentativi di aggirare il checkpoint da parte di un attacker saranno per lo più resi vani, a meno che non sia possibile confezionare un dato maligno all'interno di un valore accettato dal filtro.

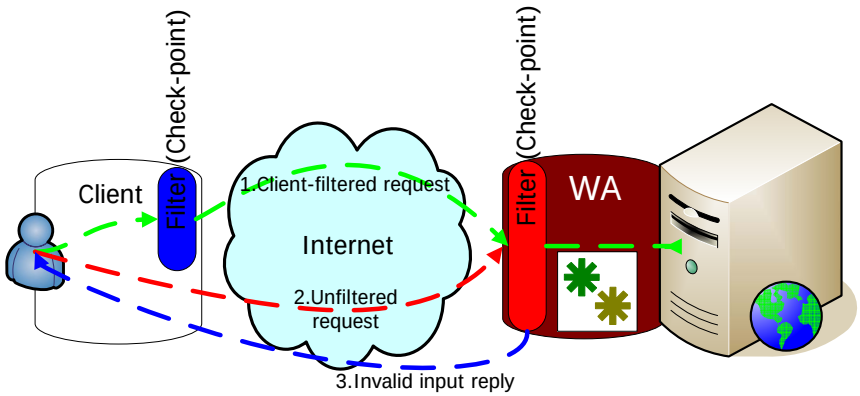

Figura 3.1: Esempi di checkpoint lato client e server

La dislocazione dei punti di controllo deve essere il più possibile completa, senza appesantire eccessivamente le prestazioni della WA: è logico e conveniente validare i dati in ingresso e uscita dei moduli componenti il nostro sistema, non ogni volta che questi vengono utilizzati. Una volta passato un controllo, e una volta entrato in una zona *safe* della nostra architettura, il dato non può essere più corrotto.

I due diversi approcci presentati si traducono, in termini di analisi sulle stringhe all'interno del codice, in due modi diametralmente opposti di affrontare il problema. Se consideriamo una generica WA, estraniando tutti gli elementi nel codice che non riguardano le stringhe, possiamo generalizzare la struttura dell'applicazione con il semplice schema presentato in Figura 3.2.

Dopo aver definito delle stringhe, tramite varie *String Definition*, (staticamente o dinamicamente attraverso parametri in input), la logica applicativa elabora queste informazioni durante diverse fasi *String Operation*. Queste

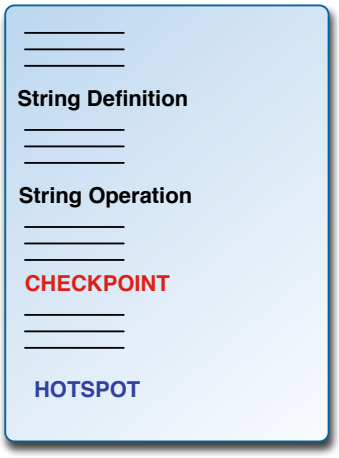

Figura 3.2: Analisi nel caso di blacklist o whitelist

trasformazioni, che possono ovviamente susseguirsi lungo tutto il corpo dei metodi, comportano la modifica e l'elaborazione delle stringhe per scopi propri della WA; esempi di metodi Java utilizzati spesso in questa fase sono *concat()*, *toLowerCase()*, *toUpperCase()*, *trim()*, *length()* e così via. Ad un certo punto del codice incontreremo poi i noti checkpoint implementati secondo una logica di tipo blacklist o whitelist, ed infine il metodo potenzialmente pericoloso (hotspot). Lo sviluppo dei checkpoint secondo i due approcci cambia notevolmente i risultati della nostra analisi, in virtù dei diversi meccanismi implementativi che i programmatori moderni utilizzano.

Nel caso di blacklist infatti si ricorre a classici meccanismi di *escaping* e di sostituzione dei caratteri che materialmente vengono implementati usando le classiche funzioni *replace()*, *subString()*, etc. Questi metodi, lavorando nel dominio delle stringhe, saranno oggetto della nostra analisi; riusciremo quindi a considerare questi checkpoint in maniera molto efficace, escludendo problematiche di generazione di falsi positivi.

Nel caso di whitelist invece gli sviluppatori ricorrono spesso a validazioni molto differenti, che escono dal raggio di azione del nostro strumento. Vengono infatti spesso usate soluzioni che utilizzano espressioni regolari e *type checking* su oggetti molto diversi dai classici oggetti *String*. In tutti questi casi il nostro strumento potrà incappare inevitabilmente in falsi po-

sitivi. Fortunatamente le applicazioni che utilizzano questi approcci sono tendenzialmente più sicure e, comunque sia, il rischio maggiore sarà quello di numerose segnalazioni associate a righe di codice in realtà sicure. Un fase di auditing manuale, assistito dal nostro strumento, permetterà comunque di discriminare queste casistiche da reali problemi di sicurezza.

3.5.2 Variabili e linguaggio associato

La metodologia proposta si basa sulla ricerca di punti di potenziale pericolo, d'ora in poi denominati *hotspot* e sulla valutazione del linguaggio associato ad ogni variabile pericolosa utilizzata per l'invocazione del metodo.

Con il termine hotspot indichiamo quei punti in cui si eseguono operazioni i cui parametri possono divenire veicolo di attacco: si tratta di operazioni intrinsecamente pericolose (metodi di output, query verso database, gestione del filesystem e così via).

Il nostro obiettivo sarà quello di ricostruire il contenuto delle variabili usate all'interno di questi hotspot, ad un certo punto dell'esecuzione del programma, e comparare il linguaggio associato ad un'opportuno linguaggio "safe" associato al metodo potenzialmente vulnerabile. Come abbiamo precedentemente illustrato, la costruzione dell'approssimazione del valore contenuto in ogni singola variabile dovrà tenere conto degli eventuali checkpoint inseriti dal programmatore.

Al termine dell'analisi vera e propria ci preoccuperemo poi di segnalare all'utente la presenza di eventuali vulnerabilità indicando in maniera puntuale dove risulta necessario verificare e sistemare manualmente il codice.

Parlando delle variabili all'interno dei metodi pericolosi abbiamo parlato di linguaggio e non di singolo valore: una variabile, all'interno di una istanza di esecuzione, può contenere un solo valore (associazione iniettiva). Se però consideriamo l'insieme delle infinite possibili prove di esecuzione di un'applicazione, automaticamente otteniamo infinite relazioni di associazione, il che significa che la nostra variabile a seconda del contesto potrà assumere valori diversi: all'interno della prova n-esima potrà assumere il valore n-esimo, e così via. Generalizzando si può dire che, interpretando le infinite istanze di esecuzione di un programma come un unico processo che

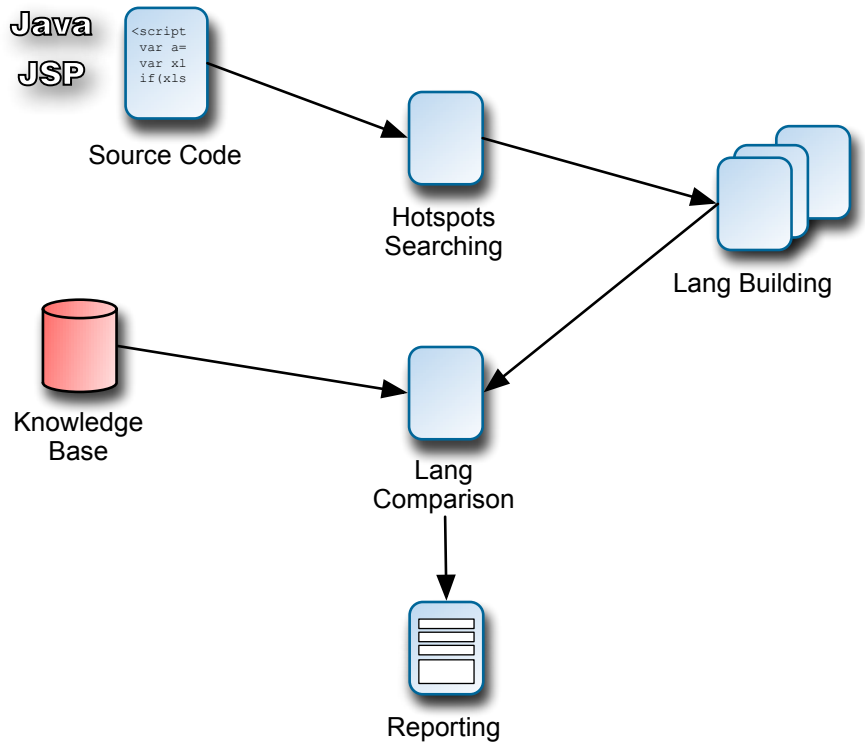

Figura 3.3: Panoramica sul metodo proposto

gestisce infiniti sotto-processi, parallelamente ad ogni variabile del codice potrà essere associato un set al più infinito di valori (finito, ad esempio, nei casi di variabili che ricevono assegnamenti costanti). L'astrazione che dobbiamo usare per trattare queste variabili non è quindi il singolo valore, ma il *linguaggio* ad esse associato, definito appunto come l'insieme dei valori potenzialmente assumibili in infinite prove di esecuzione. Posta questa condizione, deriva la considerazione che le variabili di nostro interesse vengono trattate a livello *testuale*: ci concentreremo quindi sulle operazioni, e sui checkpoint, implementati manipolando dati di tipo *java.lang.String* e *java.lang.StringBuffer*.

In fase di comparazione tra il linguaggio "safe" presente nella nostra base di dati e quello costruito dobbiamo considerare i casi in cui quest'ultimo sia

contenuto nel primo; posto L_d il linguaggio che supponiamo essere corretto, e L_b il linguaggio costruito, occorre avere:

$$L_b \subseteq L_d$$

affinchè possa essere scartata l'ipotesi di vulnerabilità.

Capitolo 4

Costruzione dei linguaggi

Dopo aver introdotto sommariamente il metodo di analisi che proponiamo, vogliamo illustrare nel dettaglio gli aspetti teorici legati alla costruzione del linguaggio, associato ad ogni singolo parametro, presente nei metodi potenzialmente vulnerabili. Per migliorare la comprensibilità e fornire una visione generale del metodo inizieremo questo capitolo con una panoramica riassuntiva.

4.1 Sintesi del metodo

L'analisi che vogliamo studiare cerca di ricostruire il valore associato ad ogni singola variabile, di tipo stringa, presente all'interno del codice sorgente. L'esatta valutazione di questo tipo di problematiche è certamente indecidibile, ma d'altra parte è possibile garantire l'approssimazione ad un valore conservativo che contenga sicuramente tutti i casi assunti realmente in fase di esecuzione. L'analisi di programmi tramite tecniche statiche per la determinazione di approssimazioni sui valori ottenibili a run-time risulta in effetti computazionalmente possibile [55]; solo recentemente però sono state proposte tecniche di analisi efficienti e sufficientemente precise per la determinazione dei valori nelle variabili di tipo stringa [56] che si distaccano dalle tecniche standard (*abstract interpretation* [57], *set constraints* [58]).

Solitamente, in questo genere di metodi, si costruisce per prima cosa il *flow graph* relativo al programma sotto analisi, per poi analizzare tale

struttura simbolica; nel nostro caso tramite analisi sul flow graph cercheremo di identificare gli automi a stati finiti corrispondenti alle approssimazioni dei valori stringa. In dettaglio:

- Partiremo dalla descrizione astratta del programma analizzato sotto forma di flow graph, in cui ogni singolo nodo del grafo rappresenta la creazione di una stringa oppure operazioni su di esse; per "operazioni su stringhe" intendiamo tutti i metodi associati a variabili di tipo *java.lang.String* e *java.lang.StringBuffer*

- Convertiremo la rappresentazione flow graph del programma in una grammatica *Context-Free* o *BNF* dotata di particolari proprietà per agevolare le successive manipolazioni e per migliorare la precisione dello strumento. Tale grammatica libera dal contesto definisce per ogni non terminale i possibili valori delle espressioni stringhe nei corrispondenti nodi del grafo

- La grammatica *BNF* verrà trasformata in una grammatica *non-self-embedding* (NSE) usando una variante dell'algoritmo Mohri-Nederhof [59]

- Selezioneremo le variabili candidate di cui vogliamo calcolare l'approssimazione del linguaggio regolare. Per lo studio del linguaggio regolare associato ad ogni hotspot ci affideremo ad un formalismo conosciuto come MLFA (*multi level finite automaton*), un grafo diretto aciclico (DAG) di automi a stati finiti non deterministici

- Per ogni singolo hotspot, passeremo poi alla rappresentazione in forma di automa deterministico minimo e genereremo, se necessario, l'espressione regolare soggiacente [60]

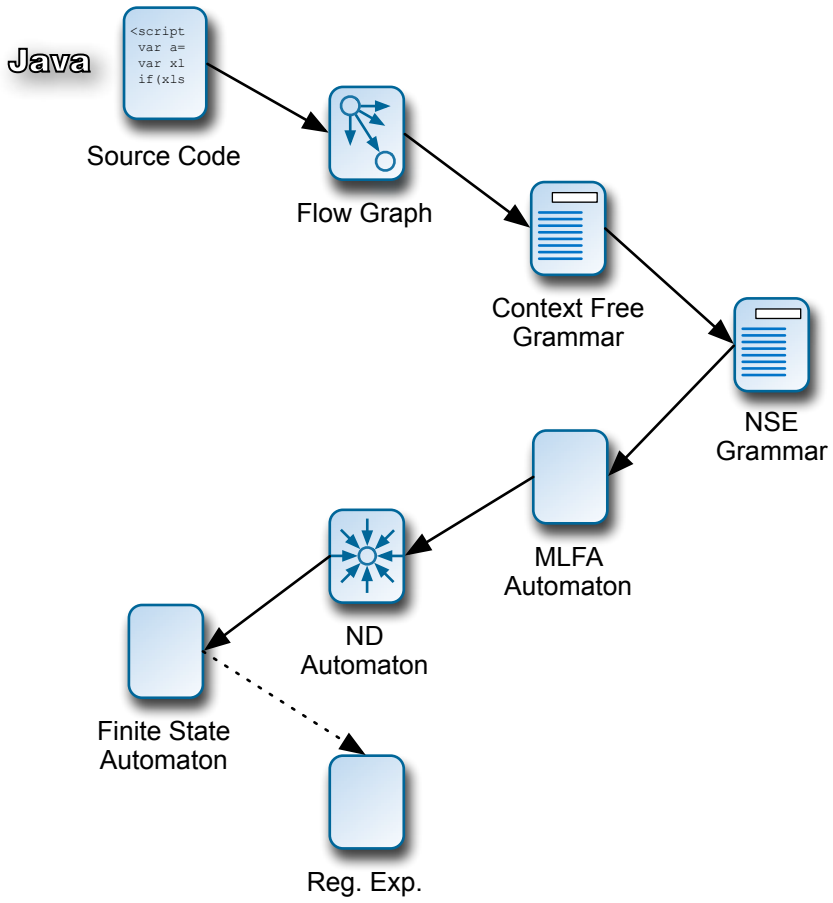

Figura 4.1: Panoramica sul percorso di analisi per la ricostruzione delle stringhe

4.1.1 Notazioni

Nel seguito del capitolo approfondiremo in dettaglio concetti relativi ad automi, grammatiche e grafi; per omogeneità del testo è quindi necessario introdurre le notazioni formali che utilizzeremo.

I linguaggi regolari e liberi che considereremo sono definiti usando come caratteri terminali quelli appartenenti all'alfabeto *Unicode* che denoteremo con il carattere Σ. I linguaggi regolari verranno opportunamente rappresentati tramite delle espressioni regolari; tale formalismo si qualifica come strumento valido e flessibile per il riconoscimento dell'input nelle applicazioni [61].

Nel seguito utilizzeremo la seguente struttura sintattica [62]:

```
regexp ::= unionexp
unionexp ::= interexp | unionexp (union)
             |      interexp
interexp ::= concatexp & interexp (intersection) [OPT]
             |      concatexp
concatexp ::= repeatexp concatexp (concatenation)
             |      repeatexp
repeatexp ::= repeatexp ? (zero or one occurrence)
             |      repeatexp * (zero or more occurrences)
             |      repeatexp + (one or more occurrences)
             |      repeatexp {n} (n occurrences)
             |      repeatexp {n,} (n or more occurrences)
             |      repeatexp {n,m} (n to m occurrences, including both)
             |      complexp
complexp ::= ~ complexp (complement) [OPT]
             |      charclassexp
charclassexp ::= [ charclasses ] (char class)
             |      [^ charclasses ] (negated char class)
             |      simpleexp charclasses ::= charclass charclasses
             |      charclass
charclass ::= charexp - charexp (chars range)
             |      charexp
simpleexp ::= charexp
             |       . (any single character)
             |       # (the empty language) [OPT]
             |       @ (any string) [OPT]
             |       " <Unicode string without double-quotes> " (a string)
```

```
|       ( ) (the empty string)
|       ( unionexp ) (precedence override)
|       < <identifier> > (named automaton) [OPT]
|       <n-m> (numerical interval) [OPT] charexp ::= <Unicode>
        (a single non-reserved character)
|        \ <Unicode character> (a single char)
```

Per la definizione delle grammatiche nel testo utilizzeremo la convenzione che comporta l'uso di lettere latine maiuscole per i caratteri non terminali, mentre i caratteri terminali saranno scritti senza particolari accorgimenti; l'alfabeto terminale e non terminale saranno disgiunti. Per quanto riguarda le rappresentazioni grafiche degli automi a stati finiti ci rifaremo alle normali convenzioni dei diagrammi *stato-transizione* in cui l'automa è rappresentato da un grafo, i nodi del grafo sono gli stati dell'unità di controllo mentre gli archi mostrano i cambiamenti di stato (le transizioni).

Per la rappresentazione del linguaggio associato agli hotspot utilizzeremo un recente formalismo [56] denominato *multi level finite automaton* (MLFA) che consiste in un insieme finito di stati Q ed un set di transizioni $\delta \subseteq Q \times T \times Q$ dove T è un set di etichette tra le seguenti: reg, ϵ, (p, q), $op_1(p, q)$, $op_2((p_1, q_1), (p_2, q_2))$ in cui p e q sono stati di Q. A queste etichette sono associati diversi linguaggi in accordo con la nomenclatura:

- $\bar{\mathscr{L}}(reg) = [reg]$
- $\bar{\mathscr{L}}(\epsilon) = \{\epsilon\}$
- $\bar{\mathscr{L}}((p, q)) = \mathscr{L}(p, q)$
- $\bar{\mathscr{L}}(op_1(p, q)) = [op_1]_R(\mathscr{L}(p, q))$
- $\bar{\mathscr{L}}(op_2((p_1, q_1)(p_2, q_2))) = [op_2]_R(\mathscr{L}(p_1, q_1), \mathscr{L}(p_2, q_2))$

In questo formalismo esiste inoltre il concetto di livello l: $Q \rightarrow \mathbb{N}$ definito come:

- $(s, (p, q), t) \in \delta \Rightarrow l(s) = l(t) > l(p) = l(q)$
- $(s, op_1(p, q), t) \in \delta \Rightarrow l(s) = l(t) > l(p) = l(q)$

- $(s, op_2((p_1, q_1), (p_2, q_2)), t) \in \delta \Rightarrow l(s) = l(t) > l(p_i) = l(q_i)$ per $i = 1, 2$

dove gli stati menzionati all'interno delle etichette di transizione sono sempre ad un livello inferiore rispetto allo stato di arrivo. Questa particolare struttura gode di due importanti proprietà:

- Una grammatica *non-self-embedding* (NSE) può essere trasformata in un MLFA in un tempo lineare rispetto al numero di regole della grammatica stessa.

- Dall'MLFA è possibile estrarre in maniera efficiente un tradizionale automa deterministico minimo.

4.2 Flow graph

La fase iniziale del metodo consiste nella costruzione del *flow graph* relativo al programma sotto revisione. Questo genere di diagramma, usato per l'analisi delle stringhe, è in grado di catturare le definizioni e le operazioni su questo genere di variabili, astraendo ogni altro aspetto all'interno del flusso di esecuzione del programma. I nodi rappresentano infatti variabili o espressioni su stringhe, mentre gli archi rappresentano la direzione del flusso delle informazioni. Più precisamente è possibile definire il flow graph utilizzato nella nostra analisi come un grafo composto da un numero finito N di nodi, ciascuno appartenente ad una delle seguenti categorie:

Init : rappresenta il costruttore di un oggetto stringa (*java.lang.String* e *java.lang.StringBuffer*) a cui è associata una particolare espressione regolare *reg* che denota il linguaggio regolare associato alla stringa. Le stringhe possono essere inizializzate tramite costanti o tramite metodi *toString()*

Join : assegnamenti e altre operazioni di join

Concat : concatenazione tra stringhe

UnaryOp : operazioni su stringhe con un solo operatore, che denotano funzioni da $\Sigma^* \rightarrow \Sigma^*$; gli argomenti della funzione, diversi da stringhe, sono considerati come semplici simboli associati alla funzione stessa

BinaryOp : operazioni su stringhe con due operatori, che denotano funzioni da $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

Per definizione, i nodi di tipo **Init** non hanno nessun arco in ingresso, nodi di tipo **Join** hanno un numero arbitrario di rami in ingresso, nodi **UnaryOp** hanno uno e un solo arco in ingresso mentre nodi di tipo **Concat** e **BinaryOp** hanno due rami in ingresso di cui è necessario rispettare l'ordinamento. Risulta quindi chiaro come, in ogni nodo n di un grafo così costruito, sia presente il valore di un'espressione stringa all'interno del programma in un certo punto n del flusso di esecuzione. La costruzione del flow graph per un programma Java è nella peggior ipotesi un'operazione quadratica rispetto alle dimensioni del programma stesso, anche se per molti casi applicativi la conversione risulta lineare. Per assicurare che il linguaggio finale contenga sicuramente il valore assunto a run-time, sono fissate le seguenti regole:

- $F(n) \supseteq [reg]$ Per ogni nodo n di tipo **Init**
- $F(n) \supseteq F(m)$ Per ogni ramo da un nodo m ad un nodo n di tipo **Join**
- $F(n) \supseteq F(m)F(p)$ Per ogni nodo n di tipo **Concat** con rami da m a p
- $F(n) \supseteq [op_1](F(m))$ Per ogni nodo n di tipo **UnaryOp** con rami da m
- $F(n) \supseteq [op_2](F(m), F(p))$ Per ogni nodo n di tipo **BinaryOp** con rami da m e p

dove $F(n)$ associa al nodo n tutti i possibili valori contenuti nelle espressioni del programma sotto analisi.

Se consideriamo il seguente spezzone di codice:

```
1 public class SimpleTest{
2
3   String strOp(String s){
4    s = s.trim();
5    s = s.replace("c","a");
6    s = s.substring(1);
7    String d = "c";
8    s = d.concat(s);
9    //result: cba
10   return s;
11  }
12
13  public static void main(String args[]){
14   String temp="abc";
15   System.out.println(new SimpleTest().strOp(temp));
16  }
17
18 }
```

Il flow graph associato sarà quello mostrato in Figura 4.2.

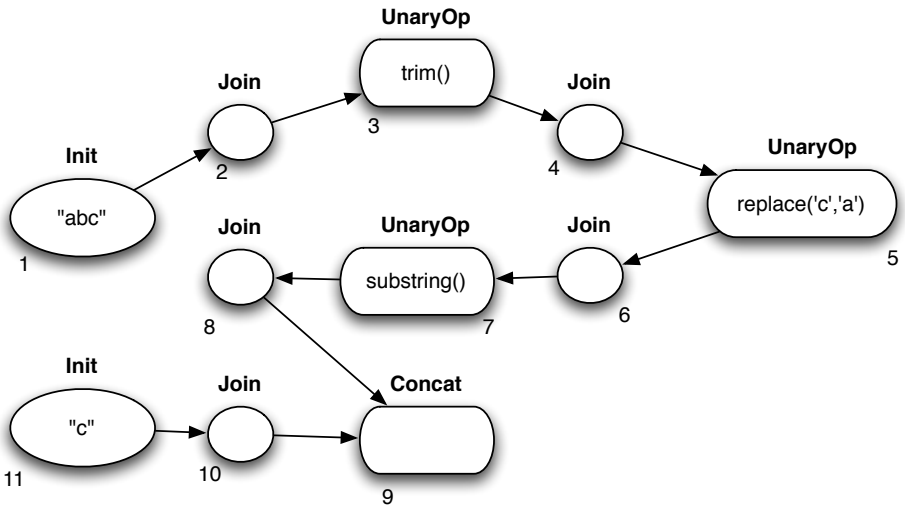

Figura 4.2: Flow graph diagram della classe Java *SimpleTest*

4.3 Costruzione della grammatica Context-Free

Dopo aver costruito il flow graph del programma, ci occupiamo ora di trasformarlo in una grammatica *Context-Free*, in maniera che ogni nodo $n \in N$ sia associato ad un elemento non terminale della nostra grammatica. Questa particolare grammatica contempla l'uso di produzioni del seguente tipo:

- $X \rightarrow Y$ [unit]
- $X \rightarrow YZ$ [pair]
- $X \rightarrow reg$ [regular]
- $X \rightarrow op_1(Y)$ [unary operation]
- $X \rightarrow op_2(Y, Z)$ [binary operation]

e determina un linguaggio in cui nella produzione $X \rightarrow reg$, X deriva tutte le stringhe componibili tramite espressioni regolari, nella produzione $X \rightarrow op_1(Y)$, X compone tutte le stringhe derivabili dall'operazione unaria applicata su tutti i possibili non terminali associati a Y. La trasformazione dal flow graph alla grammatica si attua aggiungendo un carattere non terminale A_n per ogni nodo n ed un insieme di produzioni correlate agli archi in ingresso in n. Se in n entra un nodo di tipo **Init** si aggiunge una produzione $A_n \rightarrow reg$; per un nodo **Join** si aggiunge una produzione $A_n \rightarrow A_m$ per ogni nodo m che entra in n; per un nodo di tipo **Concat** si aggiunge una produzione $A_n \rightarrow A_m A_p$; per un nodo di tipo **UnaryOp** si aggiunge una produzione $A_n \rightarrow op_1(A_m)$ ed infine per un nodo di tipo **BinaryOp** si aggiunge una produzione $A_n \rightarrow op_2(A_m, A_p)$.

Per l'esempio Java *SimpleTest* otteniamo la seguente grammatica, in cui ogni X_n indica un diverso non terminale associato al nodo n-esimo del grafo.

$X_1 \rightarrow abc$

$X_{11} \rightarrow c$

$X_{10} \rightarrow X_{11}$

$X_2 \rightarrow X_1$

$X_3 \rightarrow trim(X_2)$

$X_4 \rightarrow X_3$

$X_5 \rightarrow replace[c, a](X_4)$

$X_6 \rightarrow X_5$

$X_7 \rightarrow substring[1](X_6)$

$X_8 \rightarrow X_7$

$X_9 \rightarrow X_{10}X_8$

4.4 Dalla grammatica al linguaggio regolare

Giunti a questo punto dell'analisi vogliamo cercare di trasformare la grammatica BNF in una grammatica regolare che quindi possa generare stringhe appartenenti ad un linguaggio dello stesso tipo. L'approccio, seguito da [56] e ripreso anche durante la nostra analisi, fa uso di un concetto introdotto da uno dei padri fondatori dei linguaggi formali: Chomsky [63], il quale dimostra come una grammatica *non-self-embedding* (NSE) generi linguaggi regolari o razionali. Poichè generazione e riconoscimento di linguaggi formali sono problematiche duali ma equivalenti [64], è sempre possibile passare da un algoritmo di enumerazione (grammatica) ad uno di riconoscimento (automa), in modo completamente meccanico.

Nel dettaglio, si utilizzerà una versione leggermente modificata dell'algoritmo denominato *Mohri-Nederhof* [59]; nella versione originale dell'algoritmo la grammatica BNF viene trasformata direttamente in una grammatica *non-self-embedding* con un solo non terminale nella parte destra delle produzioni mentre qui considereremo ancora grammatiche non-self-embedding, ma che possono contenere uno o più non terminali. Nel nostro caso quindi dovremo passare da questa rappresentazione agli automi MLFA, prima di poter operare con i tradizionali automi a stati finiti. La trasformazione della grammatica BNF implica delle approssimazioni sulla grammatica stessa che devono essere considerate attentamente, in quanto possono deteriorare la qualità dei risultati. Come illustrato nell'algoritmo originale, per prima

cosa si determinano i componenti (sottografi) che sono *fortemente connessi* all'interno di un grafo costruito nel seguente modo: ogni non terminale è rappresentato da un nodo, ogni produzione da un arco che collega il non terminale sinistro alla parte destra della regola.

Per l'esempio *SimpleTest*, non avendo componenti fortemente connessi, otteniamo il grafo di Figura 4.3.

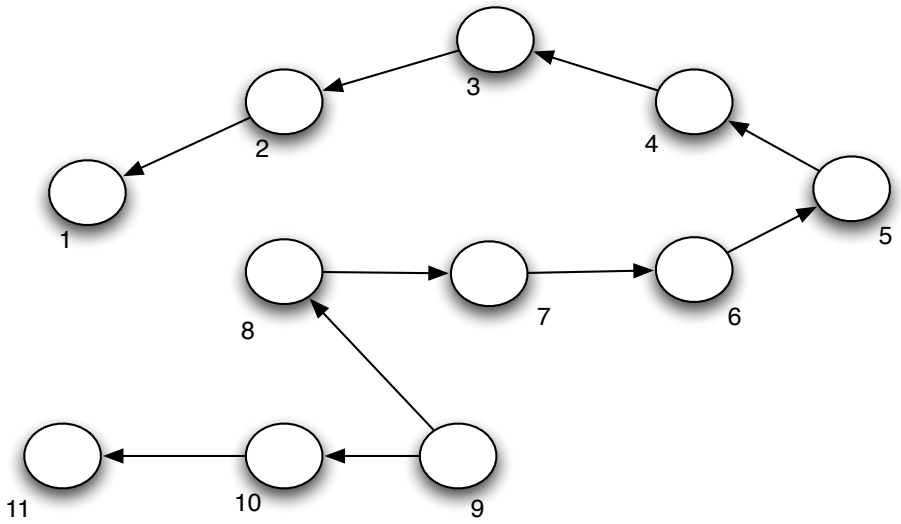

Figura 4.3: Grafo usato per l'approssimazione regolare della classe *SimpleTest*

L'algoritmo di approssimazione *Mohri-Nederhof* può essere applicato solamente in tutte le produzioni in cui i non terminali nella parte destra delle regole appartengono a componenti diversi rispetto ai non terminali nella parte sinistra; non devono quindi esistere ricorsioni dirette o indirette. Per eliminare questi cicli, inseriremo eventualmente una produzione del tipo $X \rightarrow r$ dove r denota il linguaggio regolare $C(X)^*$. Indichiamo con $C(X) \subseteq \Sigma$ un insieme che contiene tutti i caratteri che possono apparire nel linguaggio di X.

Dopo questa prima fase, considerando nuovamente i componenti fortemente connessi, possiamo catalogare questi elementi in quattro diverse categorie: **Semplice**, **Lineare sinistro**, **Lineare destro**, **Non fortemente**

regolare. Questa divisione viene fatta a seconda che siano o meno componenti generati a destra o a sinistra. Definiamo un componente M come generato a destra se esiste una produzione $A \rightarrow BC$ con A e B appartenenti ad M; equivalentemente a sinistra, se esiste una produzione $A \rightarrow BC$ con A e C appartenenti ad M.

Una grammatica BNF è *fortemente regolare* se non ha componenti non fortemente regolari, in accordo con la convenzione di *Mohri-Nederhof*; trasformando questi elementi "estranei" risulta possibile definire una grammatica Context-Free ristretta che sicuramente genera linguaggi regolari. La classe delle grammatiche fortemente regolari coincide con quelle denominate *non-self-embedding*. Per chiarezza, richiamiamo la definizione di grammatica *self-embedding* (SE):

Definizione 1 *Una grammatica Context-Free $G = (V, T, P, S)$ dove V, T, P, S sono rispettivamente l'alfabeto non terminale, l'alfabeto terminale, l'insieme delle produzioni, l'assioma è self embedding (SE) se, per ogni non terminale A, $A \stackrel{*}{\Rightarrow} \alpha A \beta$ con α e $\beta \in (V \cup T)^+$.*

Conseguentemente, una grammatica risulta *non-self-embedding* (NSE) se per ogni A, $A \stackrel{*}{\Rightarrow} \alpha A \beta$ con α oppure β uguale a ϵ.

Nel nostro caso queste produzioni non fortemente regolari M verranno convertite in produzioni lineari destre: ogni non terminale A in M comporterà l'aggiunta di un nuovo non terminale A' e, nel caso A corrisponda ad un hotspot o sia usato in altri componenti di M, si aggiungerà una produzione $A' \rightarrow \{\epsilon\}$ dove ϵ denota il linguaggio formato dalla sola ϵ-mossa.

Per le produzioni aventi A nella parte sinistra:

$$A \rightarrow X \longmapsto A \rightarrow XA'$$

$$A \rightarrow B \longmapsto A \rightarrow B, B' \rightarrow A'$$

$$A \rightarrow XY \longmapsto A \rightarrow RA', R \rightarrow XY$$

$$A \rightarrow XB \longmapsto A \rightarrow XB, B' \rightarrow A'$$

$$A \rightarrow BX \longmapsto A \rightarrow B, B' \rightarrow XA'$$

$$A \rightarrow BC \longmapsto A \rightarrow B, B' \rightarrow C, C' \rightarrow A'$$

$$A \to reg \longmapsto A \to RA', R \to reg$$

$$A \to op_1(X) \longmapsto A \to RA', R \to op_1(X)$$

$$A \to op_2(X, Y) \longmapsto A \to RA', R \to op_2(X, Y)$$

dove con A, B e C sono considerati dei non terminali in M, mentre X e Y non terminali al di fuori di esso. Per costruzione, il linguaggio associato ad un hotspot nella grammatica originale è sempre un sottoinsieme del linguaggio dello stesso non terminale nella grammatica approssimata. Per ogni operazione unaria op_1 e binaria op_2 viene usata un'approssimazione conservativa in maniera che una volta sostituite le operazioni, con le rispettive controparti approssimate, il linguaggio associato ad ogni non terminale sia regolare.

Nell'esempio Java *SimpleTest* non abbiamo nessuna componente fortemente connessa che quindi può risultare *non fortemente regolare*; l'algoritmo di approssimazione lascia la grammatica 4.3 invariata.
Per esemplificare il processo di modifica della grammatica secondo l'algoritmo di Mohri-Nederhof, riportiamo l'approssimazione della classica grammatica delle espressioni aritmetiche.

$$E \to E + T$$

$$E \to T$$

$$T \to T * F$$

$$T \to F$$

$$F \to (E)$$

$$F \to a$$

Che diventa la seguente grammatica:

$E' \to \epsilon$ $\quad$ $E \to T$

$T' \to \epsilon$ $\quad$ $T' \to E'$

$F' \to \epsilon$ $\quad$ $T \to T$

$E \to E$ $\quad$ $T' \to *F$

$E' \to +T$ $\quad$ $F' \to T'$

$T' \to E' \qquad T \to F$

$F' \to T' \qquad E' \to)F'$

$F \to (E \qquad F \to aF'$

4.5 Automi

La grammatica fortemente regolare può quindi essere agilmente trasformata in un automa MLFA come definito nel paragrafo 4.1.1. Per ogni non terminale A viene costruito uno stato q_A; per ogni componente fortemente connesso M viene costruito uno stato q_M; per ogni componente M vengono poi aggiunte delle transizioni a seconda della tipologia di M e della presenza o meno di parti sinistre in M.

Per le componenti che abbiamo chiamato **Semplici** e **Lineari destre**:

$A \to B \longmapsto (q_A, \epsilon, q_B)$

$A \to X \longmapsto (q_A, F(X), q_M)$

$A \to XB \longmapsto (q_A, F(X), q_B)$

$A \to XY \longmapsto (q_A, F(X), p), (p, F(Y), q_M)$

$A \to reg \longmapsto (q_A, reg, q_M)$

$A \to op_1(X) \longmapsto (q_A, op_1(F(X)), q_M)$

$A \to op_2(X, Y) \longmapsto (q_A, op_2(F(X), F(Y)), q_M)$

Per le componenti **Lineari sinistre** invece abbiamo:

$A \to B \longmapsto (q_B, \epsilon, q_A)$

$A \to X \longmapsto (q_M, F(X), q_A)$

$A \to BX \longmapsto (q_B, F(X), q_A)$

$A \to XY \longmapsto (q_M, F(X), p), (p, F(Y), q_A)$

$A \to reg \longmapsto (q_M, reg, q_A)$

$A \to op_1(X) \longmapsto (q_M, op_1(F(X)), q_A)$

$$A \rightarrow op_2(X, Y) \longmapsto (q_M, op_2(F(X), F(Y)), q_A)$$

in cui con $F(X)$ abbiamo rappresentato una funzione che mappa ogni non terminale X in una coppia di stati: se A appartiene ad una componente M di categoria **Semplice** o **Lineare destra**, allora $F(A) = (q_A, q_M)$, altrimenti $F(A) = (q_M, q_A)$. Questo procedimento che sembra così complesso in realtà effettua la semplice conversione di una grammatica lineare destra o sinistra in un automa, struttura formale che risulta implementativamente più semplice e performante. Giunti a questa forma, un *hotspot* presente nel codice sorgente del programma viene trasformato da semplice nodo n nel flow graph ad elemento non terminale A_n, sino ad essere associato ad una coppia di stati (s, f) nell'automa MLFA. Dalla coppia di stati, in un automa MLFA, è possibile estrarre un automa non deterministico il cui linguaggio sia $\mathscr{L}(s, f)$, attraverso il seguente algoritmo:

- Per ogni stato q nell'automa MLFA dove $l(q) = l(s)$, costruisco uno stato q' nell'automa tradizionale. Inoltre, s' sarà lo stato iniziale mentre f' quello finale.

- Per ogni transizione (q_1, t, q_2) nell'automa MLFA dove $l(q_1) = l(q_2) = l(s)$ costruisco un automa da q'_1 a q'_2 a seconda del tipo di t:

- $t = reg \Longrightarrow$ automa con linguaggio associato $[reg]$

- $t = \epsilon \Longrightarrow$ automa con l.a. $[reg]$

- $t = (p, q) \Longrightarrow$ automa con l.a. $\mathscr{L}(p, q)$

- $t = op_1(p, q) \Longrightarrow$ automa con l.a. $[op_1]_R\mathscr{L}(p, q)$

- $t = op_2((p_1, q_1), (p_2, q_2)) \Longrightarrow$ automa con l.a. $[op_2]_R\mathscr{L}((p_1, q_1), (p_2, q_2))$

Avendo ormai a che fare con i tradizionali automi, possiamo adottare una serie di tecniche classiche [64] per rendere deterministico l'automa (costruzione dell'automa deterministico con l'insieme delle parti finite) e per minimizzarlo (tramite analisi delle classi di equivalenza [65]). Poichè a livello teorico sappiamo che per i linguaggi di questo tipo il riconoscitore deterministico minimo rispetto al numero degli stati è unico, a meno di isomorfismi,

conviene ridurre gli automi che analizziamo in maniera da ottimizzare la gestione della memoria nell'implementazione software.

4.6 Dai metodi Java ad operazioni sull'automa

All'interno dell'analisi abbiamo posto particolare attenzione sulle approssimazioni introdotte poichè, come già accennato, volevamo assicurare la costruzione di linguaggi che risultassero "conservativi" rispetto al valore effettivamente assunto dalle variabili a run-time. Attraverso il metodo seguito si assicura che, se un programma Java in un certo punto della sua esecuzione, produce una particolare stringa, allora questa stringa sarà sicuramente accettata dall'automa riconoscitore costruito in quel medesimo punto del programma. Per garantire questa caratteristica del metodo teorico, le operazioni su stringhe non sono trattate a livello di flow graph ma come operazioni su automi; le trasformazioni vengono eseguite a questo livello, semplificando l'elaborazione di una struttura complessa come quella del linguaggio associato ad ogni hotspot. È quindi necessario *trasporre* le operazioni che operano sulle variabili *java.lang.String* e *java.lang.StringBuffer* in operazioni sugli automi. Il nostro scopo, come già mostrato, è quello di ottenere un nuovo automa $A(L')$ a partire da uno iniziale $A(L)$, applicando la trasformazione $T(f)$ dipendente dalla funzione f eseguita nello spazio delle variabili. L'azione che stiamo definendo è esattamente $T(f)$.

$$A(L) \overset{T(f)}{\longmapsto} A(L')$$

All'interno del codice Java dei nostri programmi analizzati dovremmo quindi tener conto di tutte le operazioni su stringhe [66] [67]; il set di metodi che consideriamo comprende:

StringBuffer.delete(int,int)

StringBuffer.deleteCharAt(int)

StringBuffer.insert(int, java.lang.Object)

StringBuffer.substring(int)

StringBuffer.substring(0,int)

String.replace(char,char): in cui entrambi i caratteri sono noti

String.replace(char,char): in cui solo il primo carattere è noto

String.replace(char,char): in cui solo il secondo carattere è noto

String.replace(char,char): in cui nessun carattere è noto

StringBuffer.replace(int,int, java.langString)

StringBuffer.reverse()

StringBuffer.setCharAt(int,char): in cui il carattere è noto

StringBuffer.setCharAt(int,char): in cui il carattere non è noto

StringBuffer.setLength(int)

StringBuffer.substring(int,int)

String.toLowerCase()

String.toUpperCase()

String.trim()

Per illustrare il processo di trasformazione da operazioni su variabili stringhe a trasformazioni nello spazio degli automi, useremo i seguenti metodi estratti dalla classe *java.lang.String*:

***concat(java.lang.String a)*:** permette il concatenamento di una stringa passata come parametro a quella su cui viene invocato il comando, e restituisce il concatenamento delle due stringhe

***trim()*:** elimina gli spazi iniziali e finali della stringa su cui viene invocato, restituendo il risultato in una nuova stringa

***toLowerCase()*:** crea e ritorna una nuova stringa, i cui caratteri sono identici a quella sulla quale è invocato il metodo, ad eccezione di quelli maiuscoli che vengono convertiti in minuscolo

***toUpperCase()*:** restituisce una stringa i cui caratteri sono identici a quella sulla quale è invocato il metodo, ad eccezione dei caratteri minuscoli che vengono convertiti in maiuscolo

***replace(char a, char b)*:** restituisce una stringa uguale a quella su cui viene invocato, nella quale le occorrenze del carattere a sono sostituite da b

4.6.1 Metodo *concat()*

Il metodo di concatenamento (o la somma algebrica di stringhe) viene risolto in modo banale, costruendo un automa che sia il concatenamento ordinato dei due automi corrispondenti ai linguaggi associati agli argomenti nello spazio delle stringhe. Otterremo, di conseguenza, un automa il cui stato iniziale sarà equivalente a quello del suo primo "addendo", mentre il suo stato finale sarà quello del secondo termine del concatenamento. L'ultima transizione del primo termine punterà allo stato iniziale del secondo, ottenendo così un automa riconoscitore del concatenamento dei due linguaggi da sommare.

Ricordiamo, infatti, che dati due linguaggi L_a e L_b, possiamo definire il linguaggio L_c (concatenamento) come:

$$L_c = L_a L_b = \{xy | x \in L_a, y \in L_b\}$$

La correttezza di questa operazione è fondamentale, poichè molto usata all'interno di script server-side in cui, a parti statiche presenti nel codice, si concatenano parametri forniti come input dagli utenti. Definiremo perciò il nostro automa risultante come:

$$A(L_a L_b) = A(L_a)A(L_b)$$

Per fornire un esempio, introduciamo un semplice automa che riconosce il linguaggio associato all'espressione regolare [a-z][[a-z]|[A-Z][[A-Z]|[0-9]]] (esempi di stringhe riconosciute sono: aa, ab, yh, gTT, hY2, ...).

Volendo concatenare il precedente automa, con quello mostrato nella Figura 4.5.

Otteniamo l'automa di Figura 4.6.

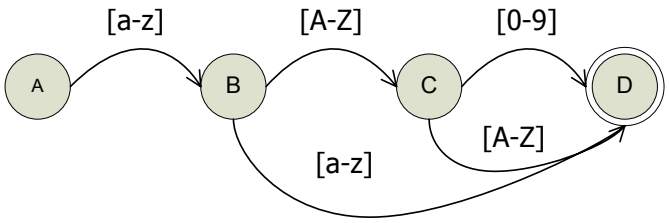

Figura 4.4: Automa per il riconoscimento di [a-z][[a-z]|[A-Z][[A-Z]|[0-9]]]

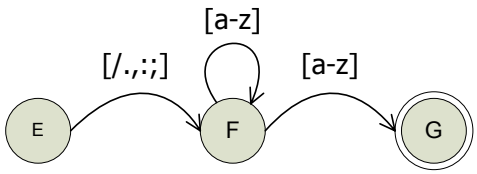

Figura 4.5: Automa da usare per l'operazione *Concat()*

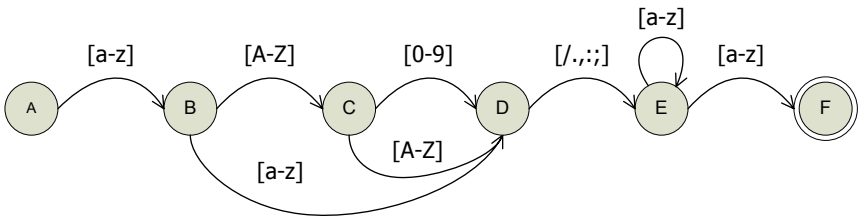

Figura 4.6: Automa risultante dopo l'operazione *Concat()*

4.6.2 Metodo *trim()*

L'operazione *trim()* causa la rimozione degli spazi di tipo *leading* o *trailing*, ovvero inseriti in testa o in coda alla stringa su cui il metodo è invocato. Una volta rintracciato l'automa associato al parametro che chiama la funzione (passaggio indispensabile per ogni *azione*), dobbiamo modificarlo in modo concorde al *significato* che il metodo ha nello spazio delle stringhe.

Concettualmente si tratta di eliminare, dai set di caratteri che consentono le transizioni, i caratteri che il metodo *trim()* rimuove, con una strategia di questo tipo: partendo dallo stato iniziale, se la transizione corrente ha consentito la rimozione di (almeno) un carattere, allora si prosegue alla transizione successiva, e in caso contrario si ferma il processo di eliminazione.

Partendo dallo stato finale, successivamente, si ragiona allo stesso modo ma percorrendo le transizioni a ritroso, partendo da quella che collega lo stato finale a quello ad esso precedente. Concettualmente, viene eliminata qualsiasi possibile catena *ininterrotta* di transizioni di caratteri *blank* partendo dal primo o dall'ultimo stato dell'automa.

Otterremo un automa $A(L_t)$, posti L_t il linguaggio dopo l'esecuzione del metodo *trim*, L_i il linguaggio iniziale, B l'insieme dei caratteri 'vuoti' e L_b l'insieme delle parole costruite su B:

$$L_t = \{xyz | xyz \in L_i; x, z \notin L_b\}$$

$$L_b = \{x_1 x_2 ... x_n | x_1, ... x_n \in B\}$$

4.6.3 Metodi *toLowerCase()* e *toUpperCase()*

Per tradurre questa operazione dobbiamo considerare ogni transizione, e trasformare il set di caratteri che rende possibile lo spostamento di stato, nel caso in cui esistano occorrenze di caratteri maiuscoli.

Formalmente, con notazione simile a quanto visto fino ad ora, avremo un automa $A(L_l)$, posti L_l il linguaggio *lowercase*, L_i il linguaggio dell'automa di partenza, e L_U il linguaggio costruito sull'insieme dei caratteri maiuscoli:

$$sL_l = \{x_1 x_2 ... x_n | x_1, x_2, ..., x_n \in L_i \wedge x_1, x_2, .., x_n \notin L_U\}$$

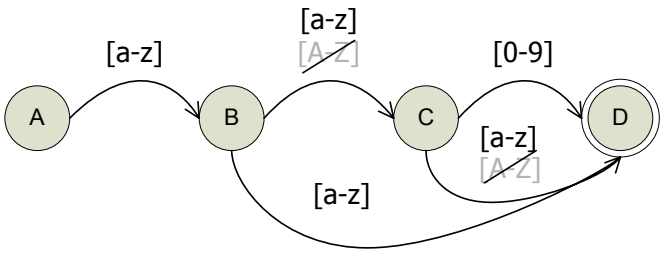

Figura 4.7: Automa risultante dopo l'operazione *toLowerCase()*

Equivalentemente, nel caso della funzione *toUpperCase()* dobbiamo svolgere la medesima trasformazione, sostituendo questa volta i caratteri minu-

scoli in maiuscoli. Di conseguenza, avremo un automa $A(L_u)$, posti L_u il linguaggio *uppercase*, L_i il linguaggio dell'automa di partenza, e L_S il linguaggio costruito sull'insieme dei caratteri minuscoli:

$$L_u = \{x_1x_2...x_n | x_1, x_2, ..., x_n \in L_i \wedge x_1, x_2, .., x_n \notin L_S\}$$

4.6.4 Metodo *replace()*

Questa operazione, all'interno della nostra analisi, verrà trattata distinguendo quattro diverse possibili configurazioni, a seconda della conoscenza statica che possediamo rispetto ai due parametri formali del metodo:

- entrambi i caratteri sono staticamente incogniti
- primo carattere staticamente incognito, secondo carattere noto
- primo carattere staticamente noto, secondo carattere incognito
- entrambi i caratteri sono noti

Il meccanismo di sostituzione su una stringa prevede che un carattere (il primo argomento) venga rimpiazzato da un altro (il secondo argomento) in tutte le sue occorrenze; possiamo pensare di traslare questo comportamento, inalterato, nello spazio degli automi: dovremo cioè andare ad analizzare ogni singola transizione esistente nell'automa, e valutando il set di caratteri che la abilita dovremo alterarlo in modo da effettuare la sostituzione tra i due parametri del metodo operante su stringhe.

Nel caso in cui entrambi i parametri formali sono noti partiremo da un automa $A(L_s)$ per ottenere un automa $A(L_r)$ relativo ad un linguaggio L_r, così costruito:

$$L_r = \{x_1x_2...x_n | x_1, .., x_n \in ((L_s \cap \neg L_O) \cup L_N)\}$$

dove L_O rappresenta il linguaggio sull'alfabeto dei caratteri sostituiti e L_N quello sui caratteri di rimpiazzo.

Se solo il primo dei caratteri è noto, ovvero quello che deve essere sostituito, otterremo invece per l'automa $A(L_r)$ il seguente linguaggio L_r:

$$L_r = \{x_1 x_2 ... x_n | x_1, .., x_n \in \Sigma\}$$

L'operazione che stiamo trattando prevede la rimozione di un carattere specifico, che immaginiamo rappresentato dal singoletto $O = \{char\}$; l'alfabeto su cui possono spaziare i valori degli atomi $x_1, .., x_n$ dovrebbe quindi essere $\Sigma \cap \overline{O}$. Ma il carattere che stiamo immettendo non è specificato, e per assurdo potrebbe addirittura essere uguale a quello che intendiamo sostituire. Quindi, ogni transizione che avviene grazie al carattere "char" appartenente ad O deve essere sostituita da una transizione determinata dall'*intero alfabeto* Σ: l'unione di questi due insiemi è, ovviamente, lo stesso alfabeto originale di L_i. Ogni lettera delle parole che possiamo comporre all'interno di questo linguaggio, dunque, appartiene al suddetto alfabeto, che sostanzialmente non è alterato da questa operazione di sostituzione.

Nel caso in cui solo il secondo carattere sia noto, l'operazione consisterà nel permettere che *tutte* le transizioni avvengano anche grazie al carattere che intendiamo immettere nel linguaggio. Non potendo sapere quale sia il carattere da rimuovere, infatti, l'unica operazione che possiamo svolgere è permettere qualsiasi transizione di stato per un set di valori uguale a quello precedente, *unito* insiemisticamente al nuovo valore. Infine, come somma delle considerazioni dei casi precedenti, nella situazione in cui nessun carattere sia noto sarà sufficiente permettere ogni transizione per un set di valori corrispondente all'intero alfabeto Σ su cui l'automa originale è stato costruito, allargando di molto i vincoli dell'automa che si va a creare e incorrendo in una approssimazione che è tanto maggiore quanto più grande è l'indeterminatezza delle informazioni in nostro possesso.

Capitolo 5

Confronto tra linguaggi

Nel capitolo precedente abbiamo visto come costruire il linguaggio associato ad un particolare hotspot; resta quindi da illustrare il processo di comparazione dei linguaggi e tutte le problematiche di approssimazione che l'analisi statica studiata comporta.

5.1 Definizione degli hotspot

Con il termine *hotspot* abbiamo definito un qualsiasi metodo, all'interno del codice sorgente dell'applicazione sotto analisi, che possa servire ad un eventuale aggressore al fine di violare il paradigma di confidenzialità, integrità e disponibilità dell'applicazione stessa.

I metodi Java che possono essere ritenuti potenzialmente "pericolosi" sono tutti quelli che permettono operazioni di input/output: se mal parametrizzato, ogni metodo appartenente a questa categoria, può generare un baco di sicurezza all'interno del software. La lettura e scrittura di file può comportare problematiche di *Path Traversal*, query e funzioni di connettività verso database possono causare *SQL Injection*, il caricamento dinamico delle classi o l'esecuzione di comandi di sistema può determinare *Command Injection* e così via. All'interno della nostra analisi, associamo ad ogni hotspot una particolare *signature* che ci permetterà di identificare univocamente ogni metodo pericoloso, anche in presenza di overloading. La base di conoscenza che contiene queste signature sarà per noi fondamentale in quanto la costru-

zione del valore delle stringhe e la successiva comparazione parte proprio dalla ricerca di queste istruzioni.

Per fornire al lettore un'idea concreta sui metodi che dobbiamo considerare, presentiamo una tabella riassuntiva di alcuni metodi Java e delle relative problematiche:

Path Traversal	Java package: **java.io**
	FileReader(java.lang.String)
	FileWriter(java.lang.String)
	FileInputStream(java.lang.String)
	FileOutputStream(java.lang.String)
	File(java.lang.String)
SQL Injection	Java package: **java.sql**
	Statement.executeUpdate(java.lang.String, ...)
	Statement.executeQuery(java.lang.String)
	Statement.execute(java.lang.String, ...)
	Statement.addBatch(java.lang.String)
	Connection.prepareStatement(java.lang.String, ...)
	Connection.prepareCall(java.lang.String, ...)
Command Injection	Java package: **java.lang**
	Runtime.exec(java.lang.String, ...)
	Runtime.exec(java.lang.String[], ...)
	System.load(java.lang.String)
	System.loadLibrary(java.lang.String)
XSS, HTTP Response manipulation	Java package: **javax.servlet**
	http.HttpServletResponse.sendError(int, java.lang.String)
	ServletOutputStream.print(java.lang.String)
	ServletOutputStream.println(java.lang.String)
	jsp.JspWriter.print(java.lang.String)
	jsp.JspWriter.println(java.lang.String)
	http.HttpServletResponse.sendRedirect(java.lang.String)
	http.HttpServletResponse.setHeader(java.lang.String, java.lang.String)

Tabella 5.1: Tabella dei metodi potenzialmente pericolosi

Ricercando sistematicamente nel codice sorgente questi hotspot, e costruendo il linguaggio associato in ognuno di questi punti, siamo quindi in grado di poter valutare staticamente l'invocazione dei metodi a meno delle approssimazioni introdotte. La valutazione dell'effettiva presenza di vul-

nerabilità verrà effettuata attraverso un confronto tra automi a stati finiti secondo un processo di comparazione che illustreremo nel seguito.

Avendo già introdotto gli obiettivi del progetto, risulta ormai chiaro come lo scopo primario della ricerca sia quello di fornire un software di supporto allo sviluppatore di programmi Java; partendo da questa assunzione considereremo come sintatticamente e semanticamente valide qualsiasi istruzione, espressione e stringa all'interno del codice sorgente. Per la parte puramente sintattica i moderni ambienti di sviluppo forniscono già un supporto insostituibile al programmatore mettendolo al riparo da semplici errori di distrazione; a livello semantico consideriamo invece le istruzioni esplicitate staticamente dallo sviluppatore (dette gergalmente "hardcoded") come *intrinsecamente* sicure. Focalizzandoci sulle problematiche di validazione dell'input dovremo quindi considerare tutti i possibili vettori in input che permettono l'inserimento di valori "non trusted". Solitamente queste sorgenti di vulnerabilità sono generalizzabili in due categorie:

Dirette : parametri GET/POST, singoli valori contenuti all'interno dei cookie usati dall'applicazione, opzioni dell'header HTTP

Indirette : Informazioni salvate, ma forgiabili da un utente (informazioni contenute in database e file testuali derivate da precedenti interazioni tra client e server)

Nel caso di servlet Java, alcuni esempi di vettori di input appartenenti alla prima categoria sono i parametri di ritorno dei metodi *getParameter*, *getParameters*, *getParameterMap*, *getComment*, *getHeader*, *getHeaders*, *getCookies*, etc.

Una rappresentazione completa ed autoesplicativa del rapporto tra vettori in ingresso e hotspot è fornita dalla Figura 5.1 (fonte dell'immagine: [68])

Tramite queste diverse sorgenti, un utente malintenzionato potrebbe quindi inserire delle stringhe che, in presenza di vulnerabilità, vanno ad alterare le invocazioni dei metodi pericolosi. La possibilità di modificare le chiamate a questi metodi, oltre a rendere sfruttabile la falla, permetterebbe anche l'invocazione degli stessi con parametri che risultino sintatticamente

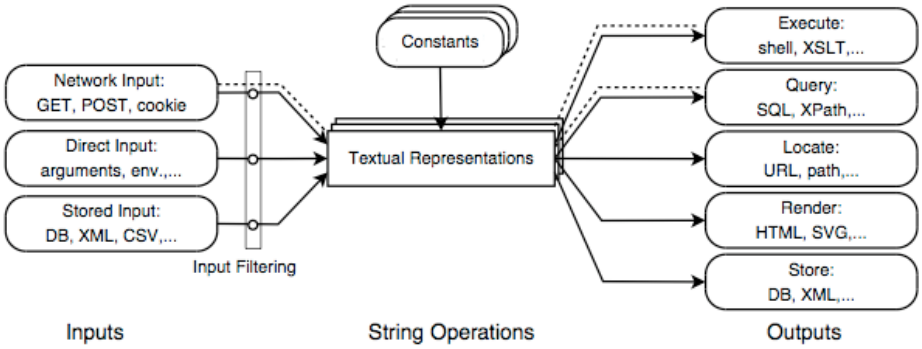

Figura 5.1: Dall'input ai parametri usati negli hotspot

errati per il metalinguaggio specifico della chiamata. Questa semplice considerazione ci permette quindi di effettuare la comparazione tra linguaggi attraverso "controesempi". Indichiamo quindi con il termine linguaggio *safe* il linguaggio che definisce delle stringhe sintatticamente valide per lo specifico metalinguaggio; l'aggettivo *safe* non deve però essere frainteso: esso si riferisce unicamente al metalinguaggio usato dalla chiamata al metodo. L'invocazione di un metodo attraverso un linguaggio *safe* sarà ritenuta insicura nel caso in cui si riesca, attraverso il processo descritto, a generare controesempi che attestino la presenza di parametri modificabili dall'utente.

Cercheremo ora di chiarire l'approccio usato attraverso un esempio pratico, in maniera da giustificare la costruzione di tali linguaggi.

Consideriamo il seguente frammento di codice:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
4 import java.sql.*;
5
6 public class Servlet extends HttpServlet {
7
8    public void doGet(HttpServletRequest request,
       HttpServletResponse response) throws ServletException,
       IOException {
9
```

```
10        response.setContentType("text/html");
11
12        try{
13         String str1= request.getParameter("param1");
14         String qry = "SELECT pass FROM table WHERE myRow ='";
15         qry = qry.concat(str1);
16         qry = qry.concat("'");
17              ...
18         Connection cn = DriverManager.getConnection(url,"user
             ","");
19         Statement cmd = cn.createStatement();
20         ResultSet res = cmd.executeQuery(qry);
21         ...
22         res.close();
23         con.close();
24       } catch (SQLException e) {
25            e.printStackTrace(); }
26   }
27   ...
28 }
```

All'interno del codice sorgente, l'istruzione *13* assegna alla stringa *str1* il valore del parametro HTTP *param1*; l'istruzione *20* invece esegue effettivamente la query contenuta nella stringa *qry* verso il database. Nel caso in cui tale query sia stata completamente specificata dal programmatore possiamo, per ipotesi, ritenere l'invocazione sicura e aderente allo standard del metalinguaggio SQL. In uno scenario, come quello presentato, la query viene dinamicamente creata in base ai parametri in ingresso immessi dall'utente; in questo caso, senza un'opportuna validazione, un aggressore potrebbe inserire istruzioni SQL (per esempio: `' OR '1'='1`) che sfruttino la vulnerabilità pur mantenendo la correttezza sintattica del parametro. Il parametro stringa *qry* rappresenta in questo caso sempre una query SQL sintatticamente corretta ma assolutamente non sicura. La possibilità di inserire istruzioni SQL permetterebbe però al malintenzionato anche di costruire query non valide. Per valutare quindi la presenza o meno di vulnerabilità all'interno degli hotspot utilizzeremo proprio questi controesempi, che invalidano la correttezza dei parametri rispetto al metalinguaggio associato.

Per determinare problematiche di SQL injection compareremo l'automa

che approssima il valore del parametro contenente la query, con il complemento dell'automa che riconosce istruzioni SQL. Poichè gran parte dei linguaggi tecnici sono stati definiti tramite grammatiche generative libere dal contesto dovremo approssimare la sintassi tramite un sottoinsieme regolare che però cerchi di non perdere completamente l'espressività e le potenzialità del linguaggio. Nel caso del linguaggio SQL, il linguaggio regolare usato copre gran parte delle query specificabili e risulta implementabile tramite un normale automa deterministico con 631 stati [69].

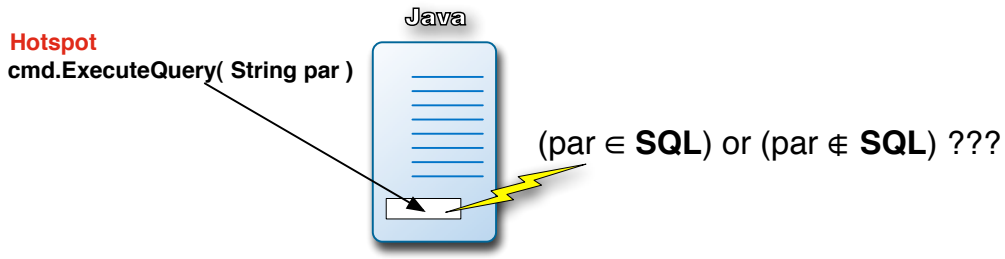

Figura 5.2: Confronto tra linguaggi nel caso di query *SQL*

Similmente, per scoprire la presenza di Path Traversal compareremo l'automa approssimante il valore run-time con quello che rappresenta qualsiasi path valido nei sistemi Windows e Unix complementato. L'automa a stati finiti che riconosce correttamente percorsi di directory validi è stato generato a partire dalla seguente grammatica libera, priva di derivazioni autoinclusive:

```
dir ::= absolutewin
      | absoluteunix
      | relativewin
      | relativeunix
absolutewin ::= AZ doubledot backslash validName* (winsep validName*)*
absoluteunix ::= slash validName* (unixpsep validName*)*
relativewin ::= validName* (winsep validName*)*
relativeunix ::= validName* (unixpsep validName*)*
AZ ::= [A-Z]
doubledot ::= :
      | %3A
      | %3a
```

```
 backslash ::= \
       | %5C
       | %5c
 validName ::= string*
       | dot string+
 winsep ::= backslash
       | dot dot backslash
 dot ::= .
       | %2e
       | %2E
slash ::= /
       | %2F
       | %2f
unixsep ::= slash
       | dot dot slash
string ::= [A-Z]+
```

Nel caso di Cross Site Scripting è invece possibile adottare due diverse politiche di confronto. Si può voler evitare l'inserimento di qualsiasi elemento appartenente al linguaggio HTML comparando ancora una volta l'automa costruito con il complemento dell'automa che approssima l'espressività del linguaggio (linguaggio che è completamente definito da una grammatica BNF). Se però, come spesso accade, l'applicazione deve consentire l'inserimento di generici tag HTML, utilizzando la comparazione precedente illustrata otterremmo un grande numero di falsi positivi. Per ovviare a tale problematica, migliorando sensibilmente i risultati forniti dallo strumento, diventa necessario costruire un automa che rappresenta il codice HTML privato dei classici tag utilizzati per l'inserimento di codice JavaScript (`<script>`, `<img>`, `<iframe>`, ...).

5.2 Confronto tra automi

Dopo aver mostrato come avviene la costruzione dell'automa per ogni hotspot e la definizione del linguaggio *safe* è necessario formalizzare il confronto tra queste due strutture formali. In particolare, posti L_b, L_d i linguaggi rispettivamente *costruito* (stimato) e *corretto* (supposto tale dalla base di

conoscenza), la comparazione deve verificare che:

$$L_b \subseteq L_d$$

per poter scartare l'ipotesi di vulnerabilità. Equivalentemente, per l'implementazione del controllo utilizzeremo lo studio dell'intersezione dei linguaggi riconosciuti dagli automi

$$(L_b \cap \neg L_d) = \emptyset$$

per determinare la presenza di metodi correttamente validati. Se il complemento del linguaggio safe risulta ad intersezione insiemistica nulla rispetto al linguaggio costruito, l'hotspot sará esente dalla vulnerabilità per cui è stato costruito tale linguaggio.

Per il tipo di contesto applicativo in cui stiamo lavorando è importante non incappare, durante l'analisi, in *falsi negativi*: errori che consistono nella mancata individuazione di vulnerabilità, quando il linguaggio stimato è più ristretto di quello reale. In questa circostanza, infatti, può accadere che il confronto con il linguaggio che supponiamo essere valido dia esito positivo, mentre se potessimo confrontare il linguaggio reale con quello valido otterremmo un esito opposto. Ne consegue che è preferibile stimare un linguaggio *più ampio* di quello effettivo, producendo al limite un *falso positivo*; errori di quest'ultimo tipo sono invece introdotti per costruzione, proprio a causa del meccanismo di sovrastima appena descritto. La conseguenza pratica di questa scelta sarà un numero maggiore di vulnerabilità rivelate, alcune delle quali potranno non essere reali. L'intento di creare una metodologia a supporto degli sviluppatori concorda perfettamente con questa scelta, lasciando al programmatore o al tester, l'eventuale verifica manuale di queste "finte" vulnerabilità. Formalmente, chiamando L_r il linguaggio *reale* (quello effettivamente formato dalle infinite prove di esecuzione del programma), si avrà un falso negativo se:

$$(L_r \not\subseteq L_d) \wedge (L_b \subseteq L_d)$$

il che implica una stima errata del linguaggio reale:

$$L_r \not\subseteq L_b$$

Si avrà un falso positivo, invece, se:

$$(L_d \subset L_b) \wedge (L_r \subseteq L_d)$$

supponendo una sovrastima corretta, ma eccessiva, di L_r.

5.2.1 Sovrastima

Parlando di falsi negativi e falsi positivi risulta indispensabile introdurre le problematiche di sovrastima del linguaggio reale associato ad una variabile di tipo stringa, all'interno di un hotspot.

Con il termine *linguaggio reale*, come accennato, indichiamo quell'insieme di stringhe teoricamente possibili, se fossimo in grado di effettuare infiniti esperimenti, come valori attuali a run-time per la variabile considerata. Se stiamo effettivamente considerando, come stima, un linguaggio più ampio del necessario, allora analizzandolo in funzione del contesto di esecuzione specifico, potremo al più rilevare errori che, disponendo del linguaggio reale, non sarebbero messi in evidenza. Stiamo, quindi, sovrastimando il *numero di rilevazioni* di vulnerabilità, incorrendo quindi in *falsi positivi*, che una fase di auditing manuale potrà, se necessario, avvallare o smentire.

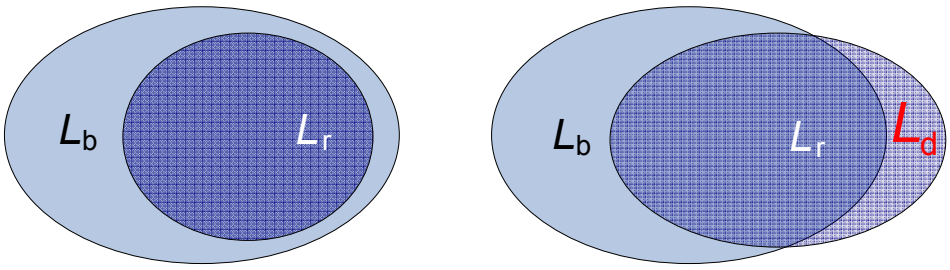

Figura 5.3: Falsi negativi durante la comparazione

Le approssimazioni introdotte dall'algoritmo di analisi, durante la costruzione dei linguaggi per gli hotspot e durante l'approssimazione dei linguaggi tecnici, devono rispettare questo requisito andando ad operare semplificazioni che non corrano il rischio di escludere dal nostro linguaggio stimato vocaboli effettivamente contenuti in quello reale. Nel caso contrario, genere-

remmo *falsi negativi*, come mostrato in Figura 5.3. Lasciando scoperta una parte L_d, le stringhe appartenenti a questo sottoinsieme potrebbero condurre ad una falla del software, senza che l'analisi basata su L_b sia in grado di riconoscere questa circostanza.

5.2.2 Correttezza

Durante la costruzione del linguaggio associato agli hotspot abbiamo cercato di mostrare la correttezza del metodo attraverso riferimenti alla teoria delle grammatiche e degli automi. Per quanto riguarda le trasformazioni degli automi, in seguito a invocazioni di metodi nello spazio delle stringhe, possiamo citare una serie di considerazioni teoriche, ben note in letteratura, che assicurano di rimanere all'interno dei linguaggi regolari partendo da automi regolari; questa proprietà è ovviamente indispensabile per poter effettuare la comparazione tra linguaggi, così come esplicitato in questo capitolo.

Formalmente le trasformazioni che operano sulla stringa in input e generano un output carattere per carattere, sono detti propriamente *omomorfismi alfabetici*; posti Σ l'alfabeto sorgente (input) e Δ l'alfabeto pozzo (output), la funzione omomorfismo è definita come:

$$h : \Sigma \rightarrow \Delta \cup \{\epsilon\}$$

e può essere semplicemente descritta da una tabella, che associ ad ogni elemento della sorgente un carattere del pozzo. Se $x = a_1 a_2 ... a_n$, con $a_1, .., a_n \in \Sigma$ e $n \geq 2$, allora potremmo calcolare la traduzione della stringa x come:

$$h(x) = h(a_1)h(a_2)...h(a_n)$$

esattamente come avviene nei nostri automi sottoposti ad azioni di trasformazione. Partiamo quindi da un automa riconoscitore del linguaggio, per poi dotarlo di una funzione di trasduzione che rappresenti il concetto di azione sulla stringa; il risultato è la costruzione di un formalismo denominato *automa trasduttore/trasformatore finito*. Grazie ad una particolare proprietà di questi automi [64]:

Proprietà 1 *La famiglia dei linguaggi regolari è chiusa rispetto alla tradu-*

zione calcolata da un automa trasformatore finito.

possiamo quindi assicurare una correttezza formale delle operazioni su stringhe nello spazio degli automi.

5.2.3 Applicabilità

Nel paragrafo 3.5.1, illustrando le possibilità fornite allo sviluppatore per la costruzione di checkpoint di validazione, abbiamo affrontato il problema della dislocazione di tali funzionalità all'interno del codice sorgente. Per la natura stessa dell'analisi che stiamo studiando non potremo mai considerare gli elementi di validazione che sono inseriti esternamente (application firewall, moduli del web server) o che utilizzano componenti di cui non possediamo il codice sorgente; librerie direttamente utilizzate all'interno del codice dell'applicazione sotto esame potranno essere incluse nella scansione, anche se spesso sono usate in formato compilato per esigenze di praticità.

L'applicabilità del metodo è quindi limitata alle WA che fanno uso di validazione estesa lungo tutto il corpo del programma o che utilizzano librerie di validazione, incluse nel progetto, in formato sorgente. Durante lo sviluppo di codice sicuro possiamo tuttavia supporre che l'uso di librerie di validazione ritenute sicure dalla comunità (quindi tendenzialmente esenti da problematiche) è spesso affiancato alla necessità di valutare le funzionalità di validazione guardando, analizzando e personalizzando il codice sorgente; in questo senso sia l'approccio basato su whitelist, che quello tramite blacklist, obbliga lo sviluppatore a dover personalizzare i propri checkpoint, lasciando spazio ad analisi di tipo statico come quella illustrata.

Nel corso della nostra metodologia abbiamo fatto riferimento alle operazioni su stringhe: molti dei metodi appartenenti alla classe stringa sono stati tralasciati poichè non operano vere e proprie trasformazioni; molti altri però non sono effettivamente considerabili durante analisi di tipo statico. I metodi che lavorano su estremi numerici, riferendosi alla posizione dei caratteri nelle stringhe, sono effettivamente difficili da trattare poichè, non potendo dare staticamente una corretta parametrizzazione, siamo obbligati a dover approssimare con il linguaggio iniziale, escludendo di fatto le trasformazioni. Nell'esempio *SimpleTest* rappresentato dal flow graph

4.2, il metodo *java.lang.String.substring(int)* verrà semplicemente ignorato, lasciando il linguaggio inalterato.

5.2.4 Fonti di approssimazione

Nella nostra analisi abbiamo spesso parlato di approssimazioni; vogliamo ora cercare di catalogare brevemente le principali fonti di imprecisione che deteriorano inevitabilmente i nostri risultati. Durante la metodologia seguita esistono infatti punti dell'analisi in cui, per ragioni teoriche o implementative, siamo costretti ad effettuare delle approssimazioni sui linguaggi o sulle operazioni; la futura ricerca su questi elementi di imprecisione potrebbe migliorare ulteriormente la bontà dello strumento.

Costruzione del linguaggio associato: approssimazione della grammatica context free per l'eliminazione delle componenti non fortemente regolari

Operazioni nello spazio delle stringhe: imprecisioni introdotte durante il mapping tra metodi su stringa e trasformazioni dell'automa

Costruzione del linguaggio *safe*: approssimazioni durante la definizione dei linguaggi safe, introdotte a causa delle trasformazione dei linguaggi tecnici in sottoinsiemi significativi ma regolari

Base di conoscenza: imprecisioni dovute alla base di conoscenza usata durante l'analisi (database delle signature, limitazioni sulla disponibilità o meno del codice sorgente)

5.3 Ricerca dei vettori

Attraverso la metodologia proposta risulta quindi possibile determinare la presenza di problematiche di input validation all'interno del codice sorgente, grazie allo studio delle stringhe negli hotspot; per gli sviluppatori, dopo aver identificato le falle, è spesso necessario determinare con precisione i vettori di ingresso che permettono lo sfruttamento delle vulnerabilità. Nel corso della nostra analisi ci siamo adoperati affinchè fosse possibile definire i possibili metodi pericolosi e, tramite comparazione dei linguaggi, definire la presenza

o meno di tali vulnerabilità, senza però mai interrogarsi sulla determinazione delle sorgenti di tali vulnerabilità. Una volta identificati puntualmente i metodi pericolosi sarebbe interessante associare a questi hotspot gli input da considerare durante il processo di validazione; queste considerazioni rappresentano tuttavia una possibile evoluzione futura al lavoro di tesi.

Una soluzione potrebbe essere la tecnica nota come *program slicing*, che ben si presta ad essere applicata all'interno del nostro contesto di analisi.

5.3.1 Slicing

Lo *slicing* nasce dall'osservazione che naturalmente i programmatori, durante la fase di test e debug, cercano in qualche modo di astrarre parti del codice sotto analisi rispetto ad un particolare punto di interesse. Una *program slice* è una "fetta" del programma principale che potenzialmente determina il valore di un insieme di variabili in un particolare punto del codice sorgente. Dalla definizione originale di Mark Weiser [70] sono state presentate molte implementazioni e tecniche ausiliarie per superare i problemi intrinseci nell'analisi statica del codice sorgente, al fine di ottenere una *slice* che risultasse corretta rispetto ad alcuni parametri e, nello stesso tempo, contenente solamente le istruzioni realmente influenti.

Decomporre i programmi in specifici blocchi, particolarmente significativi per l'analisi in oggetto, è sempre stato un approccio molto usato per far fronte alla crescente complessità dei sistemi. Anche la massima *"Divide et Impera"* è diventata, con la programmazione Object Oriented, un rimedio efficace per lo sviluppo, la manutenzione ed il testing di software. Ricavare una *slice* significa infatti ridurre un programma alle sole istruzioni che influenzano il valore di un determinato punto di interesse che viene denominato *slicing criterion*. Lo slicing criterion, nella sua accezione più classica, è definito come una coppia di valori `(numero-di-linea, variabile)`.

Lo *slicing* è quindi quella tecnica in grado di realizzare in maniera **automatica** questa riduzione del codice sorgente iniziale, evidenziando solamente le istruzioni che contribuiscono alla determinazione del valore di una specifica variabile.

Consideriamo come esempio il semplice frammento di pseudo codice presentato in Figura 5.4.

```
1 read(n);
2 i := 1;
3 sum := 0;
4 product := 1;
5 while (i <= n) do
6     begin
7     sum := sum + 1;
8     product := product + 1;
9     i := i + 1;
10    end
11 write(sum);
12 write(product);
```

Figura 5.4: Un primo esempio di *slicing*

Effettuando ora slicing con il seguente criterio ***(12, product)***, quello che vogliamo ottenere è un programma ridotto e coerente rispetto al punto di interesse considerato.

La slice sarà quindi costituita dalle istruzioni presenti nel programma precedente ad esclusione delle istruzioni alle righe **3, 7** e **11**.

Effettuare il debug dei programmi è sempre stato considerato un compito difficile: spesso la parte dispendiosa sta proprio nel determinare la sorgente del problema. Questo è dovuto al fatto che nella porzione di codice da esaminare possono esserci molte istruzioni e non tutte hanno necessariamente effetto sull'istruzione che effettivamente crea il problema. Uno strumento che calcola le program slice è un aiuto fondamentale per chi deve effettuare il debugging di un programma, anche nel caso di auditing di sicurezza. Permette al programmatore di concentrarsi solo sulle istruzioni che contribuiscono ad un malfunzionamento, rendendo più efficiente l'identificazione del punto di *fault* [71].

Weiser definisce il processo di computazione della slice usando solo informazioni statiche, senza fare quindi nessuna assunzione sull'input; la program slice S è un programma eseguibile e ridotto ottenuto da un altro programma P, rimuovendo parti di codice, in maniera da mantenere comunque una consistenza funzionale rispetto allo slicing criterion (n,V). Lo *slicing criterion*, per un particolare programma, definisce una sorta di finestra di osservazio-

ne del comportamento del codice stesso. Questa finestra è definita tramite il numero di riga (n) e il set di variabili da controllare (V). Il programma ridotto possiede quindi idealmente due caratteristiche fondamentali: è *minimale* ed *eseguibile*. Minimale poichè contiene il minor numero di linee di codice tali da rappresentare tutti i punti di interesse per lo slicing criterion: non deve esistere un'altra slice, per lo stesso criterio (n,V), con un minor numero di istruzioni.

La proprietà di minimalità è importante poichè influisce sulla reale utilità del metodo: una slice, relativamente breve, focalizza l'attenzione dello sviluppatore e del tester sulle parti fondamentali dell'algoritmo, rimuovendo quelle parti inessenziali che generano confusione e ridondanza. Purtroppo però è possibile formalmente mostrare che nessun algoritmo può sempre calcolare una slice con la minor cardinalità, in quanto non è possibile valutare l'equivalenza di due differenti parti di codice. Questa limitazione suggerisce di adattare il concetto di "dimensione della slice" specificatamente ad ogni singola analisi che si effettua. E' necessario evidenziare come il limite massimo sia comunque definito dal programma completo che può essere considerato una slice esso stesso.

La proprietà di eseguibilità è invece fondamentale per poter assicurare il corretto comportamento della slice a seguito dell'operazione di cancellazione di righe di codice dal programma originale. Il comportamento desiderabile della slice deve essere uguale a quello del programma originale, per le parti considerate e per tutti i possibili valori di input. Definire una slice sul codice significa solamente guardare lo stesso sistema (e quindi lo stesso comportamento) secondo una particolare finestra di osservazione. Questo requisito, che appare ragionevole, è però troppo difficile da soddisfare poichè il programma potrebbe anche non terminare. Gödel e Turing dimostrando l'indecidibilità della terminazione di un programma, portano Weiser a definire un vincolo meno stringente: il programma originale e la slice devono avere il medesimo comportamento se il programma originale termina. Con una slice dotata delle proprietà sopra illustrate è quindi possibile eseguire la slice e verificare il valore delle variabili considerate esattamente come se quel pezzo di codice fosse inserito all'interno di un'esecuzione che interessa tutto il programma.

Per il tipo di utilizzo delle tecniche di slicing, all'interno della nostra metodologia per la ricerca di vulnerabilità, non siamo tanto interessati all'eseguibilità della slice quanto alla sua precisione e minimalità.

Una tecnica di slicing *backward* che riesca a soddisfare efficientemente questi requisiti potrebbe quindi essere impiegata al termine della nostra analisi per risalire il grafo del programma e determinare quali punti di ingresso devono essere validati correttamente tramite checkpoint. Senza entrare nel dettaglio degli algoritmi usati [72], vogliamo solamente accennare ad un'interessante progetto che potrebbe ben integrarsi con quanto studiato.

Indus, Kaveri

Indus è un motore di slicing che effettua static slicing (backward e forward) su sorgenti Java. A livello implementativo il motore di slicing utilizza il linguaggio intermedio *Jimple*: tutti i comandi Java sono convertiti in molteplici istruzioni Jimple e la slice viene ricavata su questo nuovo set, per poi essere mappata all'indietro al termine della computazione. Indus permette di selezionare un parametro di visibilità su cui considerare la slice, definito in termini di classi, metodi o blocchi di istruzioni; inoltre permette due modalità di esecuzione differenti che privilegiano il mantenimento del valore della variabile nello slicing criterion (post-execution) secondo la definizione di Weiser [70] piuttosto che la raggiungibilità dell'istruzione definita nel criterio stesso (pre-execution). *Kaveri* è invece un front-end per l'ambiente di sviluppo Open Source *Eclipse* [52]. Data l'enorme importanza che questo ambiente di sviluppo sta assumendo negli ultimi anni e la particolare soluzione software che abbiamo implementato per dimostrare l'efficacia del metodo, sarebbe sicuramente interessante integrare le funzionalità di Indus/Kaveri, per la ricerca delle sorgenti delle vulnerabilità.

Capitolo 6

JSEC: Java.String Eclipse Checker

JSEC è l'acronimo di *Java.String Eclipse Checker*, uno strumento software che abbiamo sviluppato per validare sperimentalmente la bontà del metodo di analisi presentato in questo lavoro di tesi.

La lettera "**J**" ricorda sia la tecnologia utilizzata per implementare lo strumento, che il linguaggio su cui viene svolta l'analisi statica. Durante il design e l'implementazione di questo software sono stati affrontati tutti gli aspetti teorici, precedentemente illustrati, nel caso del linguaggio Java ed in generale della tecnologia J2EE.

Le lettere "**SEC**", oltre a ricordare la parola *security*, esplicitano la natura stessa dello strumento: un analizzatore di stringhe, plugin per il noto ambiente di sviluppo Open Source *Eclipse* [52].

6.1 Design

Nei capitoli 4 e 5 abbiamo affrontato le problematiche teoriche dell'analisi statica basata sulla comparazione dei valori di tipo stringa, all'interno di metodi potenzialmente pericolosi. Ora, vogliamo presentare il lavoro di design dell'applicazione *JSEC*, giustificando alcune scelte progettuali prese.

6.1.1 Percorso d'analisi

Lo strumento che abbiamo sviluppato si prefigge il lungimirante obiettivo di fornire un supporto efficace a sviluppatori e tester che desiderano testare le proprie applicazioni web, prima della messa in produzione del software. Molte delle scelte progettuali che hanno guidato la fase di design derivano proprio dalla volontà di creare un tool versatile per la revisione del codice, il più possibile integrato all'interno del ciclo di sviluppo del software. Possiamo rappresentare l'interazione dell'utente-tester con il semplice grafo UML di Figura 6.1.

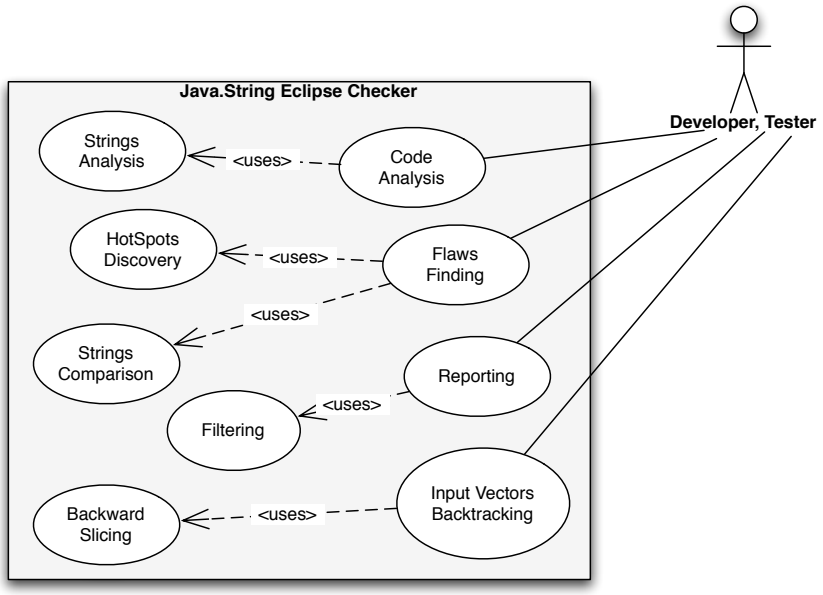

Figura 6.1: Use case: Developer, Tester

Uno scenario d'uso tipico sarà quello in cui lo sviluppatore, terminata la fase di implementazione e testing funzionale, si dedicherà all'ottimizzazione del sistema ed alla messa in sicurezza dell'applicazione. Attraverso l'interfaccia grafica del nostro strumento potrà effettuare la scansione dei file componenti il progetto, eseguire l'analisi studiata (ricerca degli hotspot, costruzione dei linguaggi, comparazione), visionare e personalizzare i risultati oltre ad effettuare analisi backward per determinare quali ingressi non

sono stati opportunamente validati. Pensando all'utilizzo del tool all'interno di grossi progetti software, è importante considerare attentamente la fase di scansione e le funzionalità di reportistica: la prima operazione deve essere quanto più ottimizzata, in maniera da velocizzare l'intero processo di analisi; per la visualizzazione dei risultati è altresì importante fornire uno strumento di reporting delle vulnerabilità trovate che sia quanto più versatile possibile. Solo riuscendo a rispettare questi requisiti uno sviluppatore potrà beneficiare della completa integrazione dello strumento, all'interno del normale ambiente di sviluppo. Lo sviluppo del nostro tool è infatti proseguita con la chiara intenzione di creare uno strumento che segua un approccio del tipo *"sviluppa, verifica, correggi"*: solo così le applicazioni future potranno essere più sicure.

6.1.2 Architettura

Partendo dalle considerazioni fatte precedentemente, dai requisiti funzionali e di presentazione, è stato realizzato, durante la fase di progettazione, il class diagram del progetto. Presentiamo ora tale schema generale dell'architettura software, mettendo in luce gli aspetti di modularità e integrazione del nostro tool; per maggiore chiarezza descrittiva considereremo singolarmente ogni package che compone il software.

All'interno del package *jsec*, rappresentato in Figura 6.2 e 6.3, sono state collocate le classi principali che gestiscono tutta la fase di scansione dei file, di analisi dei sorgenti e di rilevazione delle vulnerabilità. Per illustrare il funzionamento globale illustreremo le principali classi con una breve descrizione, evitando però di annoiare il lettore con lunghi elenchi di metodi.

Activator - Classe che si occupa di istanziare la view del plugin oltre a gestire il suo ciclo di vita, dalla costruzione sino alla disattivazione della vista all'interno dell'ambiente di sviluppo. Per migliorare lo sviluppo del plugin stesso e velocizzare la fase di debug abbiamo implementato un sistema di logging che utilizza tutta la potenza dell'ambiente di sviluppo; è possibile lanciare Eclipse in modalità debug (tramite l'opzione *-debug*) e attivare automaticamente le funzionalità di logging di *JSEC*.

Explorer - È la classe principale del tool. Si occupa di effettuare la scansione dei progetti aperti all'interno di Eclipse, determinare la locazione fisica dei file Java e dei rispettivi *.class*, istanziare i plugin disponibili, oltre ad invocare i metodi che effettuano l'analisi vera e propria. All'interno di *JSEC* utilizziamo sia la rappresentazione dei programmi in formato sorgente (java) che la rispettiva trasformazione in bytecode (class); la costruzione del flow graph e l'analisi avviene a livello di bytecode mentre il codice sorgente serve per alcune funzionalità accessorie (segnalazione delle vulnerabilità, statistiche sull'analisi, etc.). L'analisi statica effettuata a livello di bytecode non deve stupire: questa rappresentazione risulta più facilmente analizzabile da un elaboratore.

WorkPj - È la rappresentazione del singolo progetto sotto analisi. In questa classe sono contenuti attributi e metodi che servono ad identificare e a rappresentare un singolo progetto Java: nome univoco, path, numero delle classi, tempi di scansione ed analisi, lista delle vulnerabilità di tipo *SingleFlaw* presenti.

SingleFlaw - Questa classe rappresenta la singola istanza di vulnerabilità riscontrata: progetto di appartenenza, hotspot, parametro pericoloso all'interno dell'hotspot, metodo in cui è inserita l'invocazione pericolosa, numero di linea nel file sorgente, gravità della vulnerabilità, categoria, stringa pericolosa generata. Quest'ultimo parametro, in particolare, rappresenta una stringa che non fa parte del linguaggio safe e che quindi risulta generatrice di potenziali errori; almeno per l'implementazione attuale questa stringa rappresenta solamente la più semplice eccezione al linguaggio safe e non un vero e proprio exploit per la vulnerabilità in oggetto.

JFile - Rappresentazione della singola risorsa che compone il progetto. Data la necessità di gestire contemporaneamente i due livelli di rappresentazione Java (sorgente e compilato), questa classe permette di astrarre la gestione dei singoli file attraverso il concetto di risorsa.

IDetector - È l'unica classe astratta dello strumento. La possibilità, in Java, di definire delle classi parzialmente implementate risulta la so-

luzione ideale per lo sviluppo di plugin in *JSEC*. Pur derivando da questa classe, ogni plugin è dotato di funzionalità ed ottimizzazioni proprie. È all'interno di questa classe che viene effettuata la ricerca degli hotspot, la costruzione delle stringhe tramite la libreria *Brics JSA* e la comparazione dei linguaggi.

Figura 6.2: Class diagram del package *jsec*

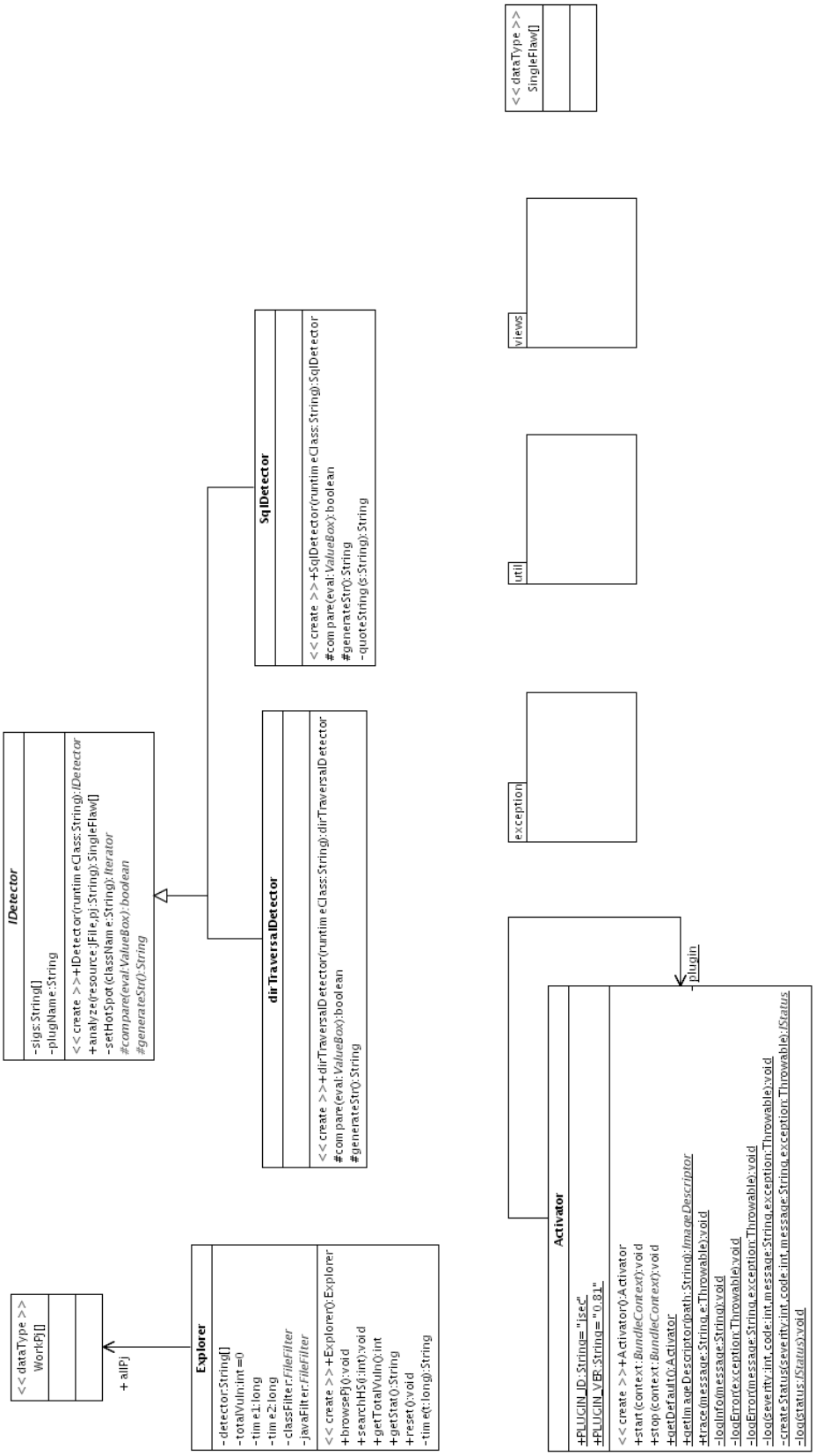

Figura 6.3: Class diagram del package *jsec*

Per l'implementazione della base di conoscenza, ci siamo affidati ad una soluzione basata su file XML al fine di rendere omogenea ed efficace la definizione e la ricerca delle signature. All'interno di tale file, chiamato *configuration.xml*, sono presenti tutte le signature degli hotspot, divise in categorie a seconda della tipologia di vulnerabilità.

Ogni signature viene rappresentata da un elemento `<hotspot>` come il seguente:

```
<!-- SQL injections -->
<hotspot id="{java.sql.Statement: int executeUpdate(String)}">
     <param>0</param>
     <category>SQL injection</category>
     <gravity>high</gravity>
</hotspot>
```

All'interno del medesimo file *configuration.xml* sono inoltre presenti le definizioni dei plugin disponibili. Per favorire lo sviluppo di plugin di terze parti, abbiamo deciso di rendere lo sviluppo e l'inserimento di nuovi detector un'operazione estremamente semplice. Sviluppare un nuovo plugin per il tool *JSEC* significa fondamentalmente considerare una specifica vulnerabilità di validazione dell'input, creare uno specifico linguaggio safe associato ai singoli hotspot, aggiornare la base di conoscenza e creare il metodo di comparazione, personalizzandolo in base al tipo di falla.

Durante la fase di testing dello strumento sono stati creati **due** plugin per il rilevamento di SQL Injection e di Path Traversal.

Per esemplificare la creazione di un plugin, illustreremo brevemente le fasi dello sviluppo:

1. Studio della vulnerabilità in oggetto. Consideriamo, per esempio, le problematiche di Path Traversal presentate alla fine del paragrafo 2.2.1

2. Analisi di tutti i possibili metodi potenzialmente pericolosi ed inserimento all'interno della base di conoscenza, secondo la notazione degli elementi *hotspot*

3. Creazione del linguaggio safe, sotto forma di automa, utilizzando una classe di servizio presente nel package *jsec.util*; nel nostro caso questo

linguaggio genera tutte le stringhe che rappresentano path validi in piattaforme Windows e Linux

4. Implementazione della semplice operazione di comparazione tra linguaggi

5. Deploy del plugin, effettuato collocando la classe nell'opportuna directory del tool ed abilitandone il caricamento dinamico tramite l'inserimento di un elemento `<detector>` nel file XML:

```
<detector id="jsec.dirTraversalDetector">
  <category>Path traversal</category>
  <regexp>dir.reg</regexp>
  <name>Path traversal Detector</name>
  <description>
  Plugin to detect Patht traversal into J2EE web apps.
  </description>
</detector>
```

La nostra implementazione realizza l'analisi sull'intero linguaggio Java, grazie all'utilizzo del framework *Soot* [73] utilizzato per il parsing dei file e la costruzione del flow graph associato al programma sotto analisi; usando questa libreria risulta possibile considerare tutti gli oggetti di tipo *String*, *StringBuffer*, oltre che gli array multidimensionali di stringhe.

Per la costruzione dei linguaggi ci siamo affidati alla libreria *Brics JSA* [74] che implementa l'approssimazione della grammatica associata al flow graph. Questa libreria, rilasciata con licenze free, è risultata molto efficiente e precisa; l'utilizzo di una tecnica chiamata *null-pointer-analysis* permette di limitare notevolmente la proliferazione di stringhe nulle, migliorando sensibilmente i risultati. L'ottima documentazione ha permesso agilmente la modifica di alcuni metodi, al fine di personalizzare ed adattare la libreria per i nostri scopi; in particolare è a questo livello che abbiamo modificato le azioni di trasformazione degli automi, rispetto ai metodi nello spazio delle stringhe, come illustrato nel paragrafo 4.6.

Per la costruzione degli automi e la gestione degli stessi abbiamo utilizzato la libreria *Brics Automaton* [62]. Attraverso questo package risulta decisamente semplice creare ed effettuare operazioni su automi a stati finiti (DFA/NFA) implementati sull'alfabeto dei caratteri terminali *Unicode* (UTF16). Oltre alle classiche operazioni (concatenazione, unione, stella di Kleene, ...), questa libreria permette azioni che sono risultate molto utili durante la nostra analisi (intersezione, complemento).

Nel package *jsec.util*, come illustrato nella Figura 6.4 sono contenute tutte le classi accessorie per l'analisi.

IOUtil - Questa classe si occupa della gestione dei file a livello di filesystem; viene utilizzata durante la scansione dei progetti ed il browsing delle singole risorse.

XmlManager - È la classe utilizzata dall'intero strumento per l'interrogazione alla base di conoscenza. Utilizzando i metodi pubblici di questa classe è possibile ottenere velocemente informazioni sui plugin e sugli hospot.

***Automaton -** Tutte le classi il cui nome termina con la parola *Automaton*, forniscono funzionalità di supporto alla creazione dei linguaggi safe. Questi linguaggi safe, da semplici grammatiche, vengono convertiti in automi binari proprio grazie a queste classi.

Nel package *jsec.exception*, mostrato in Figura 6.5, abbiamo creato due classi che estendono le classiche eccezioni Java, al fine di personalizzare l'eventuale occorrenza di errori che possono avvenire durante la scansione dei file e nel corso dell'analisi.

Infine, nel package *jsec.views* (Figura 6.6) sono contenute le classi che implementano le *views* del plugin all'interno dell'ambiente di sviluppo.

Per meglio illustrare il tipo di estensione che abbiamo sviluppato, introduciamo brevemente l'architettura della piattaforma.

Eclipse è un progetto Open Source con l'arduo intento di creare uno strumento versatile per ogni situazione all'interno del ciclo di produzione

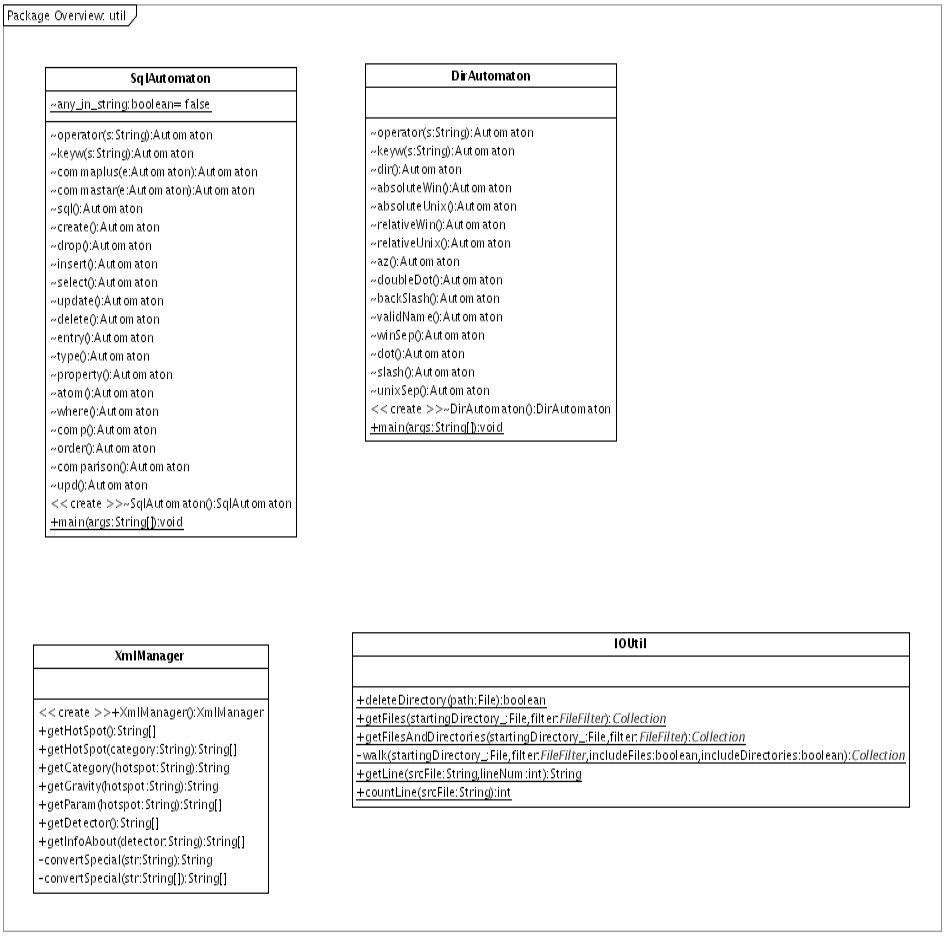

Figura 6.4: Class diagram del package *jsec.util*

del software: dalla fase di progettazione, sviluppo, debug sino al messa in produzione. La struttura completamente modulare, realizzata tramite plugin, permette di personalizzare ed adattare l'ambiente di sviluppo per ogni esigenza. All'interno dell'*Eclipse Platform* che rappresenta la vera e propria struttura portante dell'IDE è possibile distinguere due classi principali di componenti: *UI components* e *Core components*. La parte denominata *Core* definisce l'infrastruttura a plugin, gestisce il ciclo di vita, realizza l'astrazione rappresentata dal concetto di *workspace*: uno spazio di risorse (file, immagini, etc.) che corrisponde ad ogni singolo progetto sviluppato;

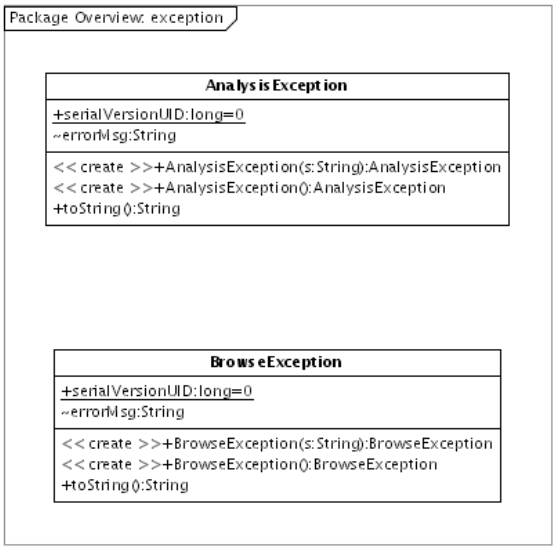

Figura 6.5: Class diagram del package *jsec.exception*

Figura 6.6: Class diagram del package *jsec.views*

nei componenti *UI* (User Interface) troviamo tutti i moduli che si occupano della gestione grafica dell'infrastruttura. Proprio all'interno di questa categoria, in particolare nel *Workbench*, viene definito il paradigma base dell'interfaccia grafica basato sul concetto di *editor*, *views* e *perspectives*.

Il plugin da noi sviluppato estende il concetto di view, creando una finestra personalizzata e realizzando così il front-end per il nostro strumento di auditing. Per la natura stessa del nostro tool non sono richieste grandi modalità di interazione tra utente e plugin, e nemmeno tra plugin e piattaforma. È solamente necessario fornire dei meccanismi per avviare, terminare e analizzare i risultati della scansione in maniera quanto più gradevole e semplificata possibile. Disponendo di un framework altamente riusabile abbiamo inoltre implementato alcune funzionalità di "copia&incolla", apertura dei file con vulnerabilità all'interno dell'editor, segnalazione degli errori tramite meccanismi propri dell'ambiente di sviluppo.

Per illustrare il risultato dell'implementazione della view mostriamo quindi alcuni screenshot dello strumento. In particolare, in Figura 6.7, è possibile vedere l'interfaccia del plugin, perfettamente integrata all'interno dell'ambiente di sviluppo. In Figura 6.8 viene invece presentata la visualizzazione dei messaggi di log, durante il debug del plugin stesso. Al termine della scansione è inoltre possibile ottenere delle semplici statistiche rispetto ai progetti analizzati e ai risultati della scansione (vedi Figura 6.9).

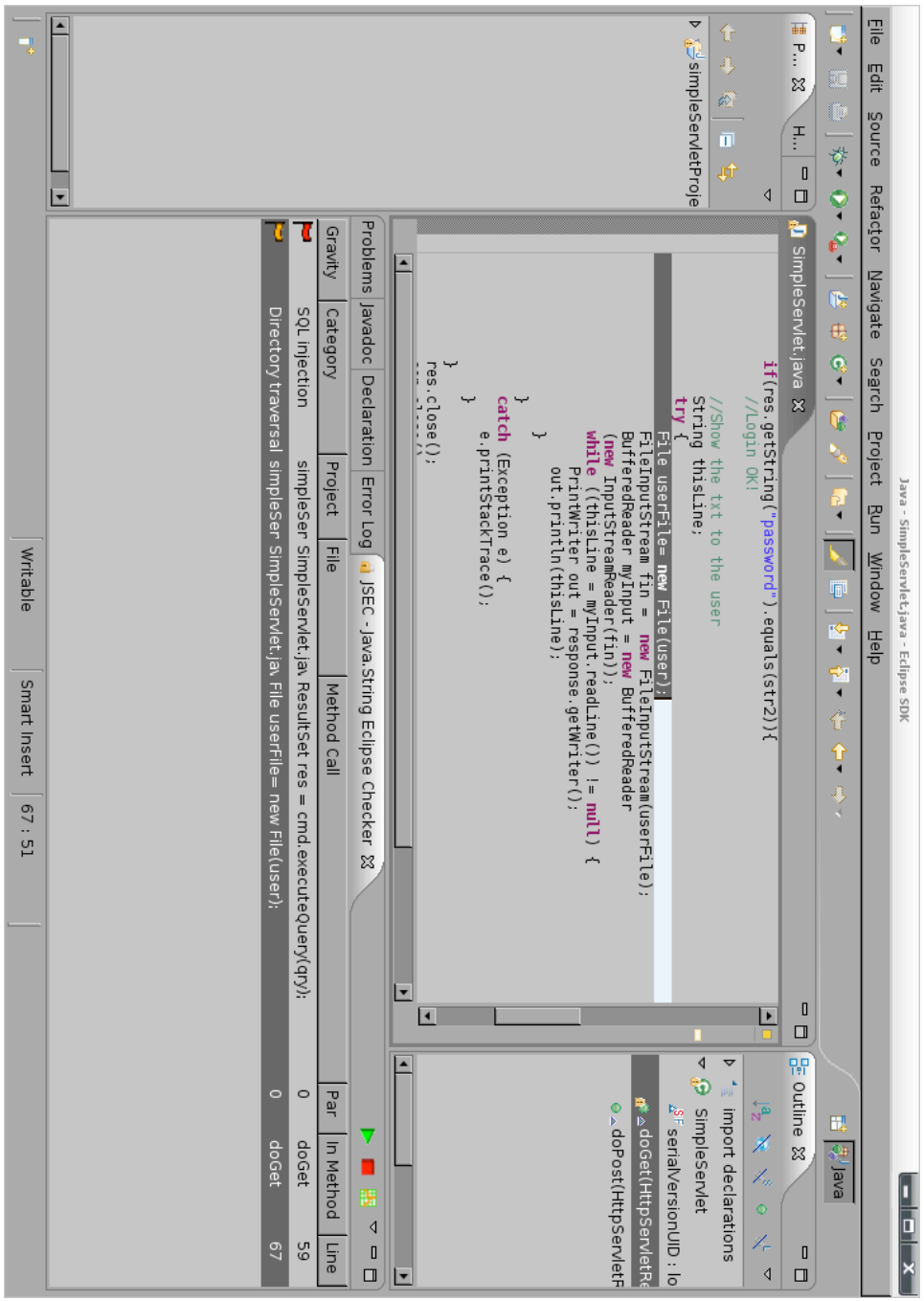

Figura 6.7: Screenshot della View *JSEC*

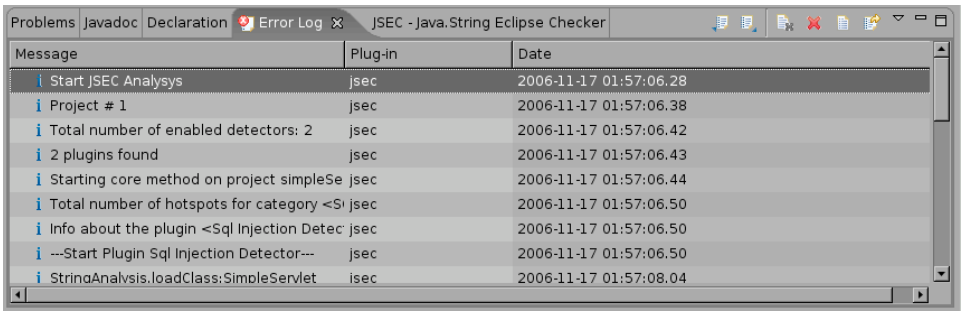

Figura 6.8: Screenshot dell'applicazione *JSEC* in modalità debug

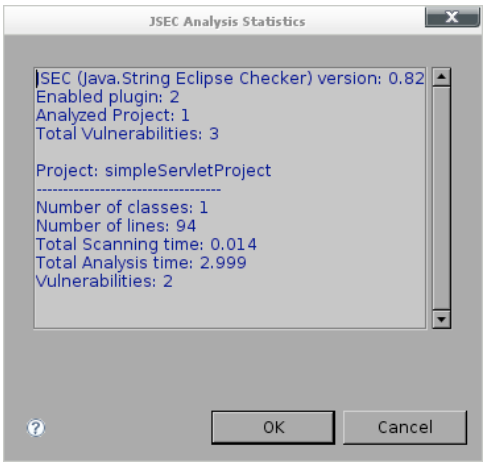

Figura 6.9: Screenshot della finestra di presentazione delle statistiche post analisi

6.2 Testing e Analisi dei risultati

Terminata la descrizione della progettazione concettuale e dello sviluppo implementativo, ci accingiamo a valutare i risultati ottenuti durante alcune analisi test.

Mentre il numero di prodotti commerciali ed Open Source per il rilevamento di vulnerabilità è aumentato notevolmente in questi ultimi anni, non esiste un approccio generale per il testing di questi strumenti e per la validazione dei risultati. Alcune soluzioni di benchmark [75, 76] per il mondo

delle applicazioni web fanno uso di applicazioni create ad hoc e contenenti vulnerabilità volontariamente inserite; altri approcci [77, 78] fanno invece uso di applicazioni reali che, dopo attenti processi di revisione del codice, sono diventati casi di test, poichè si conosce l'esatta collocazione delle vulnerabilità. In particolare, il security benchmark sviluppato dall'università di Stanford [77] contiene sette applicazioni reali e un'applicazione di test, selezionati tra i più noti progetti Open Source sviluppati in Java; particolarmente interessante la composizione delle applicazioni che spazia da progetti medio-piccoli, sino a complessi sistemi software per un totale di 200000 righe di codice. Sebbene quest'ultimo progetto sembri essere la soluzione alla verifica e alla comparazione degli strumenti di analisi statica in ambiente web, non sono ancora stati rilasciati gli elenchi ufficiali delle vulnerabilità presenti.

Ad oggi, per una valutazione scientificamente valida sul numero di falsi positivi e falsi negativi rilevati dai vari strumenti, non esistono adeguati test case.

Per la valutazione dell'efficacia del metodo e dell'implementazione del nostro strumento intendiamo, inizialmente, presentare un semplice caso di test su di una servlet sviluppata ad hoc; successivamente illustreremo il testing effettuato su WebGoat, mostrando le vulnerabilità riscontrate oltre ad alcune considerazioni sulla bontà dei risultati.

6.2.1 Validazione su *SimpleServlet*

SimpleServlet è una Java servlet che abbiamo sviluppato con l'intento di analizzare il funzionamento del nostro tool su una semplice applicazione test. Sebbene rappresenti di fatto una "toy application", le considerazioni fatte su questo esempio risultano utili per capire il funzionamento pratico dello strumento; quali genere di tipologie di vulnerabilità è in grado di determinare e quali potrebbero essere gli eventuali problemi durante la scansione.

Mostriamo ora il codice completo di questa semplice WA:

```
1 import java.io.*;
2 import javax.servlet.*;
3 import javax.servlet.http.*;
4 import java.sql.*;
```

```
5
6 public class SimpleServlet extends HttpServlet {
7
8   static final long serialVersionUID=0;
9   public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
      IOException {
10     response.setContentType("text/html");
11     String title = "Simple Request-Parameters reader";
12
13     try {
14       // Connection string
15       String urlConnection = "jdbc:odbc:myDataSource";
16
17           /*** Request Parameters ***/
18       //USERNAME
19       String str1= request.getParameter("param1");
20       String user=str1;
21       //PASSWORD
22       String str2 = request.getParameter("param2");
23
24       //Retrieve the password of the user
25       String qry = "SELECT * FROM secretTable WHERE myRow ='".
           concat(str1).concat("' AND myPass ='").concat(str2).
           concat("'");
26
27       //DBMS connection
28       Connection con = DriverManager.getConnection (
           urlConnection, "username", "password");
29       Statement cmd = con.createStatement ();
30       ResultSet res = cmd.executeQuery(qry);
31
32       if(res.getRow()!=0){
33       //Login OK!
34       //Show the txt to the user
35       String thisLine;
36       try {
37             File userFile= new File(user);
38             FileInputStream fin =  new FileInputStream(
                 userFile);
39             BufferedReader myInput = new BufferedReader(new
```

```
                InputStreamReader(fin));
40              while ((thisLine = myInput.readLine()) != null) {
41                  PrintWriter out = response.getWriter();
42                  out.println(thisLine);
43                  } }catch (Exception e) {
44                      e.printStackTrace();}
45              }
46          res.close();
47          con.close();
48             } catch (SQLException e) {
49              e.printStackTrace();}
50 }
51    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
52          doGet(request, response);
53    }
54 }
```

SimpleServlet è una semplice servlet che riceve due parametri (riga *19* e *22*) da una precedente pagina HTML; questi due parametri corrispondono, rispettivamente, a **username** e **password** dell'utente. In particolare i parametri inseriti vengono utilizzati nella riga *25* per identificare la tupla all'interno del database corrispondente all'utente, al fine di validare o meno la presenza di un dato username, con una certa password, all'interno della base di dati (riga *32*). Possiamo quindi pensare a questa servlet come una semplice pagina di autenticazione per la visualizzazione del profilo dell'utente. Successivamente infatti, il nome dell'utente viene utilizzato in *37* come identificativo per la lettura di un file statico presente sul web server; anche questo tipo di pattern di programmazione è abbastanza comune quando si vuole includere del codice HTML statico all'interno di pagine create dinamicamente.

Importando questo semplice file all'interno di un nuovo progetto di Eclipse, abilitando la visualizzazione della vista corrispondente al nostro plugin ed eseguendo la nostra analisi sul file in oggetto, otteniamo in effetti la segnalazione di due vulnerabilità appartenenti alle categorie di SQL Injection e Path Traversal. La colonna *Gravity* indica il livello di gravità della vulnerabilità scoperta, parametro che dipende staticamente dalla tipologia *Cate-*

gory di appartenenza; con *Project*, *File* e *Line* si identifica univocamente la posizione del metodo vulnerabile *Method Call* all'interno dell'applicazione analizzata; le colonne *Parameter* e *InMethod* identificano rispettivamente la posizione del parametro vulnerabile ed il metodo in cui avviene l'invocazione pericolosa.

Identifier	Gravity	Category	Project	File	Line
1	Red Flag	SQL Injection	servletPJ	SimpleServlet.java	30
2	Orange Flag	Path Directory	servletPJ	SimpleServlet.java	37

Identifier	Method Call	Parameter	In Method
1	ResultSet res = cmd.executeQuery(qry)	0	doGet
2	File userFile = new File(user)	0	doGet

Effettuando una semplice operazione di auditing manuale ci si accorge velocemente di come entrambi i parametri non siano validati in nessun punto del codice. Un aggressore potrebbe quindi inserire del codice SQL all'interno del parametro *param2* e guadagnare così l'accesso al profilo di altri utenti; per esempio con i parametri *param1*=`"admin"` e *param2*=`"aaa' OR '1'='1"` si riuscirebbe a visualizzare il profilo dell'utente *admin*.

Per quando riguarda la segnalazione relativa alla vulnerabilità di Path Traversal è interessante notare come, sebbene la vulnerabilità esista, richieda una particolare condizione affinchè risulti effettivamente sfruttabile da un aggressore. Un utente malintenzionato, con l'intento di visualizzare il file contenente gli hash delle password nei sistemi *unix (*/etc/passwd*) sarebbe costretto ad utilizzare una tecnica decisamente più complessa della precedente. Per poter leggere file, diversi dal profilo e residenti sul web server, è richiesto un account valido effettivamente registrato sul server poichè i metodi di apertura e visualizzazione dei file sono all'interno del corpo di una funzione che implica un esito positivo della precedente ricerca di tuple sul database.

Senza possedere un account non potremmo soddisfare questa condizione: inserendo le stringhe mostrate per il caso di SQL Injection otterremmo l'esecuzione dei metodi inerenti ad operazioni sul filesystem, senza però

poter manipolare opportunamente il parametro *param1*. Un abile aggressore, disponendo della possibilità di creare un account (magari tramite un normale form, come avviene spesso in Rete) potrebbe registrare un account "speciale" inserendo, durante il processo di registrazione, lo username "*../../../../etc/passwd*" ed una qualsiasi password.

A questo punto semplicemente effettuando una normale autenticazione avrebbe accesso al file richiesto, sempre a patto di conoscere la giusta collocazione del file all'interno del filesystem; attraverso una serie di tentativi, modificando la profondità del path richiesto, è comunque semplice risalire a tale informazione. Questo caso applicativo, volontariamente realizzato, dimostra come spesso sia insostituibile l'auditing manuale del codice e come strumenti simili al nostro possano effettivamente aiutare la ricerca di vulnerabilità: dalle più semplici alle più complesse.

Considerando ora la medesima servlet in cui ogni parametro utilizzato sia opportunamente validato, attraverso funzioni che effettuano operazioni su stringhe, il nostro tool non riporta nessuna vulnerabilità. Se si considerano invece particolari implementazioni dei meccanismi di validazione otteniamo dei falsi positivi: è il caso in cui all'interno del codice sorgente sono utilizzati dei meccanismi di whitelisting su oggetti diversi dalle stringhe. In Java, per esempio, gli sviluppatori possono usare la seguente soluzione per implementare un checkpoint relativo ad una interrogazione SQL:

```
1 ...
2 final String sql = "Select * from Customer where CustomerID =?";
3 ...
4 final PreparedStatement ps = con.prepareStatement(sql);
5 ps.setString(1,customerID);
6 ...
```

Utilizzando l'oggetto *java.sql.PreparedStatement* e i relativi metodi *set*()* è possibile realizzare dei meccanismi di *type checking*, sui parametri in input, molto robusti. Questo costrutto assicura che ogni dato, fornito dalla web application, verso il database sia controllato rispetto al tipo atteso (stinghe, interi e così via).

Poichè la validazione viene effettuata però su oggetti diversi dalle strin-

ghe, i controlli di questa natura esulano dagli scopi del presente lavoro di tesi.

6.2.2 Validazione su *WebGoat*

Dopo il semplice test, svolto su una singola servlet, vogliamo effettivamente testare il nostro strumento su un progetto di dimensioni paragonabili a quelle di una generica applicazione per il web. La selezione, svolta tra i software di benchmark per la sicurezza in ambito web, ha evidenziato come il candidato migliore fosse WebGoat [75].

WebGoat è un'applicazione J2EE sviluppata con l'obiettivo di illustrare le più comuni vulnerabilità presenti in ambito web. Questa WA è effettivamente strutturata come un vero e proprio corso, dotato di diverse lezioni, in cui in ognuna di esse l'utente deve dimostrare la propria abilità nello scoprire e nello sfruttare vulnerabilità software appositamente inserite. Il sistema è dotato dei meccanismi di navigazione presenti nei comuni software online, in maniera da simulare fedelmente una classica applicazione di medie dimensioni e complessità. Per fare un esempio di una tipica lezione possiamo citare quella relativa alle tecniche di SQL Injection, in cui è presente un form web utilizzato per la verifica dei numeri delle carte di credito; l'utente, nei panni dell'aggressore, deve riuscire ad immettere delle stringhe SQL per rubare tali numeri.

Sebbene le vulnerabilità siano state inserite intenzionalmente dagli sviluppatori del progetto non esistono, ad oggi, delle statistiche ufficiali sul numero e sul tipo di vulnerabilità presenti; la conoscenza di questa informazione, purtroppo necessaria per una corretta stima dei falsi negativi, non ci permetterà di definire con precisione la completezza del test eseguito. In compenso, la scelta di usare uno strumento appositamente studiato per l'exploiting di vulnerabilità, ci permetterà di testare effettivamente gli errori trovati; basterà effettuare il deploy della medesima versione del software per valutare precisamente la qualità dello strumento e la percentuale di falsi positivi.

Tutti gli esperimenti riportati in questo capitolo sono stati effettuati su un elaboratore *Intel Pentium IV 3GHz* con 1 GB di Ram, con sistema

operativo Linux. Il plugin è stato testato all'interno di un'istanza di Eclipse in modalità sviluppo, con abilitata l'opzione debug. Nel dettaglio è stata utilizzata la versione *0.82* del plugin su Eclipse *3.2.1* con *sun-jdk-1.5.0.08*; nel plugin *JSEC* risultavano abilitati i due plugin sviluppati, relativi alle vulnerabilità di SQL Injection e Path Traversal.

L'analisi su *WebGoat* è stata svolta solamente sui moduli appartenenti alle lezioni; i tempi riportati in tabella sono da intendersi in secondi.

Test	Classes	Lines	Scan Time	Analysis Time	Vuln
WebGoat	35	8474	0.032	323.184	18
WebGoat	35	8474	0.058	440.246	18
WebGoat	35	8474	0.044	201.322	18

L'analisi ha riportato la presenza di 18 vulnerabilità di cui **5** di tipo Path Traversal e **13** di tipologia SQL Injection.

Attraverso un fase di revisione manuale del codice e successivi tentativi di attacco sull'applicazione online, è possibile affermare che **6** di queste vulnerabilità risultano dei falsi positivi: **4** SQL Injection, **2** Path Traversal.

Comparazione con il tool *LAPSE*

Per confermare ulteriormente la validità del nostro strumento abbiamo voluto confrontare l'operato del nostro tool, rispetto ad uno strumento con medesimi scopi basato su di una metodologia di analisi differente. Questa seconda indagine sarà basata sulla comparazione dei nostri risultati, nelle medesime condizioni del precedente test, rispetto al tool *LAPSE* [79].

Come illustrato in 3.4.3, LAPSE è uno strumento di analisi statica, integrato in Eclipse, di recente sviluppo. Sebbene la teoria alla base di LAPSE [80] sia differente dalla metodologia proposta in questo lavoro di tesi, risulta sicuramente interessante comparare questi due strumenti così simili negli intenti.

Per i successivi esperimenti, è stata utilizzata la versione *2.5.6* di LAPSE nelle medesime condizioni operative del test precedente; per uniformità dei risultati, anche in questo caso sono state considerate solamente vulnerabilità di tipo SQL Injection e Path Traversal.

Al termine dell'analisi sono stati rilevati i seguenti risultati:

Test	Classes	Lines	Analysis Time	Vuln
WebGoat	35	8474	21.54	9
WebGoat	35	8474	24.37	9
WebGoat	35	8474	23.81	9

di cui tutte le **9** vulnerabilità sono di tipo SQL Injection.

Attraverso auditing manuale abbiamo determinato che **1** di queste vulnerabilità è un caso di falso positivo.

Riassumendo il confronto tra i due strumenti possiamo dire che entrambi hanno rilevato **6** SQL Injection effettivamente presenti nel codice dell'applicazione; *JSEC* ha poi evidenziato altre **3** vulnerabilità dello stesso tipo, non considerate dall'altro strumento; equivalentemente *LAPSE* ne ha scoperte altre **2**, relative a differenti punti nel codice sorgente. Il confronto incrociato di questi ultimi due valori può fornire una prima stima sul numero di falsi negativi generati dai due strumenti.

Solo *JSEC* ha rilevato **3** reali vulnerabilità Path Traversal.

Il tasso di falsi positivi per *JSEC* è pari al **33%** mentre per *LAPSE* è pari all'**11%**. La velocità dell'analisi è decisamente a favore di quest'ultimo strumento, a discapito però del rilevamento per alcune particolari problematiche.

6.3 Sviluppi futuri

Lo strumento *JSEC* implementa la metodologia teoricamente presentata nei precedenti capitoli, dimostrando come un approccio basato sulla determinazione statica dei valori stringa, per la ricerca di vulnerabilità, sia più che una semplice possibilità teorica. Tuttavia esistono nuove prospettive e nuove possibilità di ricerca, che possono condurre a risvolti pratici importanti per perfezionare questo strumento di revisione del codice.

Presentiamo, nel seguito, alcune possibili espansioni del progetto.

6.3.1 Perfezionamento del tool

Il tool realizzato risulta indubbiamente in uno stato di sviluppo iniziale, sebbene possa già essere utilizzato all'interno di applicazioni reali; piccoli perfezionamenti di alcune componenti del software potrebbero sicuramente migliorare la qualità delle scansioni.

Aggiornamento delle signature, linguaggi safe

È stato più volte ricordato come l'incompletezza della base di conoscenza costituisca un fattore determinante per la qualità dei risultati. In quest'ottica risulta sicuramente importante verificare le potenzialità espressive offerte dalle nuove versioni di Java, al fine di includere tutti i metodi potenzialmente pericolosi all'interno delle signature; poichè tutta l'analisi parte dalla determinazione degli hotspot, la mancata rilevazione di questi punti, potenzialmente deboli, può inficiare l'intero processo di revisione. Per quanto riguarda i linguaggi safe è invece necessario mantenere costantemente aggiornati questi elementi, al passo con l'evoluzione dei metalinguaggi e dei sistemi informatici, per non incorrere in falsi positivi e falsi negativi.

Personalizzazione e funzionalità

Nell'applicazione che proponiamo, alcune funzionalità di personalizzazione delle scansioni e di presentazione dei risultati sono ancora relegate al lavoro futuro.

Quello di cui si dovrà tenere conto, oltre alle normali caratteristiche di configurazione di qualsiasi programma, sarà la possibilità di interagire non solo con la base di conoscenza, ma anche con il sistema di report, al fine di "raffinare" i risultati ottenuti dalla scansione attraverso filtri e meccanismi di ricerca. In quest'ottica sarebbe interessante inserire anche diversi livelli di severità della scansione, che influenzino sensibilmente la ricerca e la comparazione delle stringhe negli hotspot.

Grazie all'integrazione con uno strumento di sviluppo come Eclipse, il nostro tool fornisce già le classiche funzionalità di editing e compilazione. La velocità e l'immediatezza tra rilevazione e correzione delle vulnerabilità rappresenta uno dei punti di forza degli strumenti di revisione integrati ne-

gli ambienti di sviluppo. Un eventuale miglioramento potrebbe consistere nell'implementazione di un modulo del tool che permetta l'analisi di file *.jsp* e *.war* (Web ARchive); in entrambi i casi, rispettivamente attraverso operazioni di compilazione e decompressione, sarebbe abbastanza banale trasformare queste sorgenti in progetti direttamente analizzabili da *JSEC*.

A livello di funzionalità sarebbe poi ipotizzabile l'inserimento, a discrezione dell'utente, di checkpoint all'interno del codice sorgente. Con una conoscenza abbastanza precisa dei vari linguaggi, si potrebbe pensare ad un processo di rivelazione e correzione semi automatico dei problemi riscontrati. È ben chiaro come questo genere di approccio per risolvere i problemi di validazione sono, ad oggi, ancora poco generalizzabili ed implicano una buona precisione dell'analisi statica che spesso, per ragioni prettamente teoriche, non è possibile ottenere.

Nuovi plugin

La costante evoluzione delle tecniche di attacco implica un aggiornamento costante del tool, attraverso lo sviluppo di nuovi plugin che permettano di rilevare nuove vulnerabilità. Come illustrato, lo sviluppo di plugin è un'operazione che non richiede un grande sforzo in termini implementativi; è però fondamentale considerare attentamente il linguaggio safe associato ai nuovi plugin al fine di migliorarne l'efficacia. Per utilizzare lo strumento durante il testing di software in produzione è utile ricordare la necessità di sviluppare riconoscitori di attacchi meno recenti ma sicuramente importanti che, per ovvie ragioni, non sono stati sviluppati durante questo lavoro di tesi.

6.3.2 Backward slicing

Dopo aver illustrato alcuni miglioramenti minori, discutiamo la possibilità di migliorare l'analisi attraverso l'uso incrociato di diverse tecniche. Nel paragrafo 5.3 abbiamo affrontato lo studio di una possibile tecnica per risolvere le problematiche di ricerca dei vettori di input. Determinare puntualmente la sorgente di una vulnerabilità permetterebbe di velocizzare il processo di revisione manuale e di validazione dei risultati ottenuti utilizzando il nostro

software. Le tecniche di slicing sono sicuramente uno dei principali risultati teorici che potrebbero aiutare nella ricerca di valori in input portatori di problematiche di validazione del codice.

Grazie ai recenti sviluppi del progetto Indus/Kaveri si potrebbe ipotizzare l'utilizzo congiunto dei tool, integrati all'interno di Eclipse: l'utente, dopo aver effettuato la scansione alla ricerca di vulnerabilità, potrebbe selezionare ogni hotspot effettivamente pericoloso e con semplici operazioni tracciare quale parametro in ingresso è la causa principale della vulnerabilità.

6.3.3 Verifica online

Altro interessante sviluppo per l'attività di verifica delle vulnerabilità rilevate è costituito dal testing online, svolto a posteriori dopo l'analisi statica. Studiare una possibile interazione tra i due paradigmi di rilevamento di vulnerabilità rappresenta sicuramente una frontiera interessante per la ricerca. Partendo dal linguaggio differenziale messo in evidenza dalle operazioni di confronto dei linguaggi (stimato e atteso), e grazie alle tecniche di backward slicing, potremo generare stringhe di test da "iniettare" nella WA, valutare le risposte HTTP, in modo da validare automaticamente la presenza o meno delle vulnerabilità che supponiamo esistere. Con la sola analisi statica, corriamo infatti il rischio di incorrere spesso in *falsi positivi*; una verifica diretta potrebbe aumentare il grado di confidenza delle supposizioni del nostro analizzatore. Concettualmente, ripeteremmo le operazioni che attualmente governano il funzionamento dei black box scanner, ma partendo dal grosso vantaggio di sapere dove indirizzare gli attacchi e come formulare le richieste: questo comporterebbe un aumento di efficienza notevole.

Si renderà dunque necessario un nuovo modulo, che agisca sulla base dei report ottenuti e garantisca funzionalità di connettività, formattazione di richieste HTTP e valutazione delle risposte. Quello di cui abbiamo bisogno è un bot automatico, la cui principale funzionalità sia la navigazione e l'interazione con WA evolute. Al termine dell'invio di input potenzialmente pericolosi, il software deve essere in grado di distinguere tra il verificarsi di un errore di sicurezza e una pagina valida, basandosi sugli output generati dalla navigazione di percorsi ritenuti corretti.

Nel caso di SQL Injection, la rivelazione di vulnerabilità potrebbe avvenire tramite l'esecuzione sequenziale di tre controlli. Il primo si occupa di verificare se l'hash della pagina generata dall'input potenzialmente invalido corrisponda a quello generato da una pagina ritenuta corretta; evidentemente, nel caso di pagine sicure ma contenenti testo dinamicamente generato istante per istante, questo test è destinato a fallire. Il secondo controllo è attuato tramite la ricerca, all'interno della pagina esito potenzialmente invalida, di espressioni regolari corrispondenti ad errori dei DBMS più comuni. Il terzo controllo, invece, estrae da ogni pagina esito l'albero *Document Object Model* (DOM) e cerca, all'interno dell'insieme dei DOM validi, un albero compatibile con la struttura della pagina potenzialmente invalida.

La rilevazione nel caso di Cross Site Scripting può invece essere realizzata in maniera molto più semplice: l'algoritmo può controllare che nella pagina generata non sia presente la stringa di test, oppure sia presente ma con i tag html rimossi o modificati.

Date le potenzialità di un test di questo genere sono già state considerate alcune possibili soluzioni a livello implementativo; per lo sviluppo di un crawler, con le funzionalità richieste, si potrebbe facilmente utilizzare la libreria Java *HTTPUnit* [httpunit.sourceforge.net], creando così uno strumento versatile e funzionale.

6.3.4 Supporto multi linguaggio

Un'espansione significativa dell'intero strumento potrebbe derivare dall'introduzione di una struttura che permetta il supporto multi linguaggio. Come già chiarito, le considerazioni sulla costruzione e comparazione delle stringhe potrebbero teoricamente essere applicate ai diversi linguaggi di programmazione per il web.

L'implementazione di uno strumento di questo genere sarebbe sicuramente interessante poichè fornirebbe una soluzione comune alla ricerca di vulnerabilità tramite analisi statica. Data la necessità di creare una linea di demarcazione che separa la parte del software indipendente dal linguaggio sotto analisi, rispetto a quella in cui avviene effettivamente il parsing

dello specifico linguaggio in oggetto, si potrebbe pensare di realizzare una traduzione intermedia.

Molte delle funzionalità dello strumento attuale lavorano già in un livello astratto rispetto allo specifico linguaggio: la creazione del flow graph avviene, per esempio, tramite il framework Soot che utilizza rappresentazioni intermedie per l'analisi del bytecode Java; linguaggi intermedi molto simili sono facilmente generabili, anche per linguaggi diversi da Java, con i moduli di parsing ed analisi di GCC. Effettuando un'operazione di astrazione, per ogni passo seguito dalla nostra analisi, si potrebbe in effetti ottenere uno strumento che supporti l'auditing di WA scritte in linguaggi diversi. Lo studio legato a queste tematiche, sicuramente impegnativo ma estremamente interessante, rappresenta un notevole stimolo per le future ricerche.

Capitolo 7

Conclusioni

La ricerca di vulnerabilità all'interno di applicazioni web rappresenta sicuramente una sfida per il prossimo futuro. Sempre più persone ed aziende utilizzano quotidianamente la Rete per svolgere il proprio lavoro e per trascorrere il proprio tempo libero; conseguentemente le applicazioni diventano sempre più complesse ed evolute. All'aumentare di questo interesse, aumenta proporzionalmente l'attenzione a garantire requisiti di sicurezza sufficienti per la fornitura di servizi affidabili, dotati di meccanismi robusti di autenticazione, confidenzialità ed integrità dei dati.

Negli ultimi anni, la necessità di ricercare in maniera efficiente le possibili problematiche di sicurezza all'interno dei software online ha innescato un processo di ricerca e di sviluppo di nuove soluzioni tecnologiche per la determinazione di queste falle, sin dalla prime fasi dello sviluppo del software stesso. Rilevare le vulnerabilità e correggere le applicazioni ancora prima della loro messa in produzione rappresenta un innumerevole vantaggio, in termine economici e funzionali, per le aziende e per i semplici utenti. L'analisi statica sul codice sorgente rappresenta una di queste possibili soluzioni. Sebbene lo studio di queste tecniche risale ormai agli albori dell'informatica, l'applicazione in ambito web rappresenta una nuova sfida per gli esperti ed i ricercatori.

L'elaborazione delle informazioni, all'interno delle applicazioni web, avviene prevalentemente attraverso uno scambio di dati testuali generalizzabile

attraverso il concetto di stringa. L'applicazione di note tecniche di analisi statica, affiancate a nuove metodologie di trattamento delle stringhe, ha suggerito la possibilità teorica di realizzare uno studio che abbia come obiettivo proprio la rilevazione di vulnerabilità.

Attraverso una ricerca esaustiva sul codice sorgente vengono identificati i punti potenzialmente pericolosi all'interno delle applicazioni; questi punti di potenziale pericolo sono costituiti da tutti quei metodi e funzioni, parametrizzati attraverso delle stringhe, che accedono alla risorse interne dei sistemi informativi. Partendo da questi punti deboli, l'analisi proposta cerca di ricostruire il possibile valore assunto a run-time dai parametri; questo risultato viene successivamente comparato con un valore, considerato sicuro, al fine di determinare la presenza o meno di vulnerabilità. In particolare, l'analisi studiata ricostruisce la grammatica libera dal contesto relativa al programma sotto analisi, cercando successivamente di approssimarla ad una forma regolare che permetta la trattazione tramite automi a stati finiti. Le singole operazioni su stringhe, svolte all'interno della logica applicativa del programma, vengono trattate come semplici trasformazioni di automi in maniera da ottenere un formalismo efficiente che cattura il concetto di valore a run-time. Nella fase successiva dell'analisi, questa rappresentazione viene confrontata con una base di conoscenza in maniera da poter discriminare l'effettiva occorrenza di una data vulnerabilità software.

A supporto della metodologia presentata abbiamo poi sviluppato uno strumento per la validazione sperimentale della teoria, dimostrando come la ricerca di vulnerabilità attraverso la ricostruzione delle stringhe all'interno delle applicazioni web sia una tecnica con ottime prospettive future. Lo strumento realizzato, denominato **J***ava.***S***tring* **E***clipse* **C***hecker*, permette infatti l'analisi e l'identificazione di vulnerabilità in applicazioni J2EE attraverso una comoda strutturale modulare per il supporto e l'aggiornamento rispetto alle nuove vulnerabilità e tecniche di attacco. La scelta progettuale di creare uno strumento integrato in un noto ambiente di sviluppo ha poi evidenziato come un approccio ciclico di sviluppo, testing e messa in sicurezza sia il giusto iter per assicurare alle future applicazioni l'adeguata resistenza verso attacchi informatici.

La comparazione dei risultati sperimentali, rispetto a tool esistenti su

un campione di applicazioni, ha dimostrato non solo l'efficacia della metodologia proposta ma anche le caratteristiche distintive che in alcuni contesti applicativi potrebbero farla preferire ad altre soluzioni.

Le possibilità di miglioramento della tecnica teorica e dell'implementazione software, unita alla futuribile integrazione con altre tecniche di analisi, rappresenta una sfida ed un'interessante spunto di ricerca per il futuro.

Bibliografia

[1] Simson Garfinkel and Gene Spafford. *Web security & commerce*. O'Reilly & Associates Inc., 1997.

[2] Miniwatts Marketing Group. Internet World Stats. www.internetworldstats.com/stats.htm.

[3] Network Working Group. Rfc 2616 - Hypertext Transfer Protocol – HTTP/1.1. Technical report, June 1999.

[4] Information Sciences Institute University of Southern California Defense Advanced Research Projects Agency. Rfc 793 - Transmission Control Protocol. Technical report, September 1981.

[5] Information Sciences Institute University of Southern California Defense Advanced Research Projects Agency. Rfc 791 - Internet Protocol. Technical report, September 1981.

[6] Sun Microsystems. Java Server Pages. java.sun.com/products/jsp.

[7] The PHP Group. PHP Hypertext Preprocessor. www.php.net.

[8] Microsoft Corporation. Active Server Pages. www.asp.net.

[9] Paolo Malacarne. *Java Servlet*. Apogeo, 2001.

[10] Hans Bergsten. *Java Server Pages*. O'Reilly & Associates Inc., 2000.

[11] James Goodwill. *Pure JSP–Java Server Pages: A Code-Intensive Premium Reference*. SAMS, 2000.

[12] Marty Hall. *Core Servlets and JavaServer Pages*. Prentice Hall PTR, 2000.

[13] The Open Web Application Security Project. A guide to building secure web applications and web services. Draft v2.1, 2004. www.owasp.org.

[14] The Open Web Application Security Project. The ten most critical web application security vulnerabilities, Jan 2004. www.owasp.org.

[15] David Scott and Richard Sharp. Abstracting application-level web security. In *WWW '02: Proceedings of the eleventh international conference on World Wide Web*, pages 396–407. ACM Press, 2002.

[16] David Scott and Richard Sharp. Developing secure web applications. *IEEE Internet Computing*, 6(6):38–45, 2002.

[17] Ahmed E. Hassan and Richard C. Holt. Architecture recovery of web applications. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 349–359. ACM Press, 2002.

[18] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, 2002.

[19] J.D.Meier, A.Mackman, S.Vasireddy, M.Dunner, R.Escamilla, and A.Murukan. *Improving Web Application Security - Threats and Countermeasures*. Microsoft Press, 2003.

[20] SANS. The top 20 most critical internet security vulnerabilities. www.sans.org/top20.

[21] Sverre H. Huseby. *Innocent Code*. John Wiley & Sons, 2004.

[22] Joel Scambray, David Wong, and Mike Shema. *Hacking Exposed Web Applications: Web Application Security Secrets & Solutions*. Osborne/McGraw-Hill, 2002.

[23] Kevin Spett. SQL injection. Whitepaper, 2002. www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf.

[24] Kevin Spett. Blind SQL injection. Whitepaper, 2003. www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf.

[25] Chris Anley. Advanced SQL Injection in SQL Server Apps. Whitepaper, 2002. www.ngssoftware.com/papers/advanced_sql_injection.pdf.

[26] Chris Anley. (more) Advanced SQL injection. Whitepaper, 2002. www.ngssoftware.com/papers/more_advanced_sql_injection.pdf.

[27] Ofer Maor and Amichai Shulman. Blindfolded SQL injection. Whitepaper, 2003. www.imperva.com/download.asp?id=4.

[28] Ofer Maor and Amichai Shulman. SQL injection signatures evasion. Whitepaper, Apr 2004. www.imperva.com/download.asp?id=2.

[29] The SquirrelMail Project Team. SquirrelMail. www.squirrelmail.org.

[30] The PHP Group. PEAR - PHP extension and application repository. pear.php.net.

[31] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying Cross Site Scripting vulnerabilities in web applications. In *WSE '04: Proceedings of the Web Site Evolution, Sixth IEEE International Workshop on (WSE'04)*, pages 71–80. IEEE Computer Society, 2004.

[32] Amit Klein. Cross Site Scripting explained. Whitepaper, 2002. crypto.stanford.edu/cs155/CSS.pdf.

[33] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM Press.

[34] OWASP. Owasp Italy Chapter. www.owasp.org/index.php/Italy.

[35] Moran Surf and Amichai Shulman. How safe is it out there? Whitepaper, 2004. www.imperva.com/download.asp?id=23.

[36] WhiteHat Security. WhiteHat Security. www.whitehatsec.com.

[37] Jody Melbourne and David Jorm. Penetration testing for web applications. www.securityfocus.com, Jun 2003.

[38] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, California, feb 2000.

[39] Nevin Heintze. Aliasing analysis for a million lines of C. In *ASIA-PEPM '02: Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, pages 47–49. ACM Press, 2002.

[40] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167. ACM Press, 2003.

[41] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257. IEEE Computer Society, 2000.

[42] Brian Witten, Carl Landwehr, and Michael Caloyannides. Does open source improve system security? *IEEE Softw.*, 18(5):57–61, 2001.

[43] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *WWW '03: Proceedings of the twelfth international conference on World Wide Web*, pages 148–159. ACM Press, 2003.

[44] C.Jones. *Applied Software Measurements*. McGraw Hill, 1996.

[45] Ken Ashcraft and Dawson Engler. Using programmer -written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 143. IEEE Computer Society, 2002.

[46] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[47] wiretrip.net. Whisker. www.wiretrip.net/rfp.

[48] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[49] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities, 2006.

[50] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM Press.

[51] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, August 2005.

[52] The Eclipse Foundation. Eclipse IDE. www.eclipse.org.

[53] Jeremiah Grossman. Challenges of automated web application scanning, Oct 2003. Black Hat Federal.

[54] Jeff Williams. Input Validation, AppSec 2004 NYC. OWASP, Jun 2004. www.owasp.org.

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[56] A. Christensen, A. Mller, and M. Schwartzbach. Precise analysis of string expressions, 2003.

[57] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[58] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, 1999.

[59] M. Mohri and M. Nederhof. Regular approximation of context-free grammars through transformation, 2000.

[60] V. Laurikari. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 181. IEEE Computer Society, 2000.

[61] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language, 2002.

[62] Anders Moeller. dk.bricks.automaton. BRICS research center, University of Aarhus.

[63] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2:137–167, 1959.

[64] Stefano Crespi Reghizzi. *Linguaggi Formali e Artificiali - Aspetti Sintattici*. Città Studi Edizioni, 2000.

[65] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, 2001.

[66] Sun Microsystems Inc. java.lang.String. Java 2 Platform API.

[67] Sun Microsystems Inc. java.lang.StringBuffer. Java 2 Platform API.

[68] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.

[69] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications, 2004.

[70] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[71] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Softw. Engg.*, 7(1):49–76, 2002.

[72] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[73] Soot Devel Team. Soot: a Java optimization framework. www.sable.mcgill.ca/soot.

[74] Anders Moeller. dk.bricks.string. BRICS research center, University of Aarhus.

[75] OWASP. WebGoat. www.owasp.org/index.php/OWASP_WebGoat_Project.

[76] Foundstone Inc. Hacme books. www.foundstone.com.

[77] Stanford University. Stanford SecuriBench. suif.stanford.edu/∼livshits/work/securibench/intro.html.

[78] Stanford University. Stanford SecuriBench Micro. suif.stanford.edu/∼livshits/work/securibench-micro/index.html.

[79] Stanford University Ben Livshits. LAPSE: Web Application Security Scanner for Java. suif.stanford.edu/∼livshits/work/lapse.

[80] V. Benjamin Livshits. Findings security errors in Java applications using lightweight static analysis. Work-in-Progress Report, Annual Computer Security Applications Conference, November 2004.