UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Scienze e Tecnologie dell'Informazione

Tecniche di analisi statica per la sicurezza di applicazioni web: problematiche ed implementazioni

RELATORE:
Prof. Marco Cremonini

TESI DI LAUREA DI:
Dario Battista Ghilardi
753708

Anno Accademico 2010/2011

Ringraziamenti

Un ringraziamento particolare va al Prof. Cremonini per avermi dato la possibilità di svolgere un lavoro di tesi che coincidesse con le mie passioni ed i miei interessi.

Ringrazio il Dott. Marco 'embyte' Balduzzi di EURECOM per le indicazioni all'inizio di questo di lavoro di tesi, per i continui stimoli nell'analisi e nella raccolta del materiale e per avermi introdotto senza formalismi al mondo della ricerca accademica. Se ora ho conoscenze sufficienti a saper scegliere tra un futuro accademico ed un futuro in ambito lavorativo lo devo principalmente all'esperienza di questi mesi.

Ringrazio il Dott. Andrea Manenti per tutte le risate che ci siamo fatti durante questi anni di università, dalle lezioni in cui eravamo in aula in due, ai momenti in cui attendevamo che l'aereo atterrasse alla Malpensa, alle mille volte che ci siamo dati un passaggio a vicenda per venire in università.

Un grazie speciale alla mia famiglia. L'università è senza dubbio la migliore scelta che abbia mai fatto e non sarebbe stata possibile senza i vostri sacrifici e la vostra pazienza. Mi ha permesso di crescere, spero nei prossimi anni di mettere a frutto ciò che mi avete dato la possibilità di imparare.

Ringrazio Francesca perché ha esultato e festeggiato con me quando passavo gli esami e mi ha rincuorato quando le cose non andavano come avrei voluto. Ci sei sempre stata e spero sarà così ancora per molto tempo.

Ringrazio infine tutti gli amici e tutte le altre persone che sono state al mio fianco, siete stati tutti importanti.

Grazie.

Indice

Capitolo 1

Introduzione

Solo 15 anni fa nessuno avrebbe mai pensato che Internet sarebbe diventata parte integrante della vita quotidiana di ogni individuo. Da semplice meccanismo di condivisione delle informazioni mediante pagine statiche, la rete si è evoluta fino a diventare una realtà dinamica ed interattiva, in grado di interagire con l'utente rendendo disponibile una quantità di informazioni immensa.
Il merito di tale risultato è senza dubbio da attribuire alla quantità di applicazioni web disponibili, da enciclopedie costruite in modo collaborativo a sistemi per gestire il proprio patrimonio economico da remoto, da soluzioni per acquistare e vendere oggetti a sistemi per condividere fotografie e per gestire la propria rete sociale. Non è possibile elencare tutte le soluzioni che la rete fornisce ma si può benissimo ammettere che per moltissime attività prima svolte esclusivamente offline, ora esiste un'alternativa tramite internet.

Le applicazioni web hanno però introdotto una nuova importante problematica: la sicurezza dei dati sensibili degli utenti. Diverse tipologie di vulnerabilità possono portare alla perdita di tali dati, a modifiche nel funzionamento dell'applicazione ed alla compromissione del server sul quale essa risiede. Sebbene tale rischi siano ingenti, per molteplici cause quali il rispetto delle tempistiche, l'assenza di adeguate conoscenze di sicurezza e di programmazione, questi vengono spesso sottostimati dagli sviluppatori e la presenza di vulnerabilità risulta essere tutt'altro che rara.

Al fine di ridurre il rischio, le applicazioni web vengono riviste manualmente da team di esperti in grado di identificarne le vulnerabilità in una fase successiva a quella di sviluppo, chiamata code review. La code review è un processo lungo e con alto margine di errore, spesso condizionato dal budget e dalle tempistiche.
Vi è quindi l'esigenza di strumenti in grado di rendere più efficiente il processo di code review, tra i quali è di fondamentale importanza l'analisi statica.

L'analisi statica si è diffusa negli anni '70 con l'avvento di Lint, un tool rivolto alla ricerca automatica di problematiche nel codice sorgente di programmi scritti in C. Questa tecnica ha attualmente molteplici scopi: viene utilizzata per l'evidenziazione della sintassi e per il rispetto degli standard di scrittura codice negli IDE, per la generazione della documentazione e per trovare eventuali problematiche di sicurezza nel codice sorgente.

Proprio in quest'ultimo campo l'analisi statica può rivelarsi una scelta vincente, fornendo uno strumento alternativo alla review del codice, più impreciso ma decisamente più pratico e veloce.

Nel lavoro di tesi vengono esposte in dettaglio le modalità secondo le quali l'analisi statica volta alla ricerca di vulnerabilità può integrarsi nel ciclo di sviluppo software. Viene dimostrato che l'uso dell'analisi statica durante lo sviluppo può rendere più economica la correzione di eventuali problematiche rispetto all'analisi a posteriori e può annullare la finestra di esposizione dovuta ad eventuali vulnerabilità.

Al fine di presentare degli esempi concreti si è scelto il linguaggio PHP mediante il criterio della diffusione, con lo scopo di studiarne le peculiarità nella fase di interpretazione del codice. Tale studio ha consentito l'individuazione di punti di accesso attraverso i quali implementare meccanismi di analisi statica, confermati nella successiva analisi degli approcci implementati dagli strumenti open source esistenti, quali Pixy, Saner e RIPS.
In letteratura sono presenti numerosi studi[29] che si occupano di analizzare in dettaglio i risultati che i vari strumenti di analisi statica riportano su determinate codebase, tuttavia questi effettuano un'analisi a scatola chiusa, limitandosi a trarre conclusioni in base all'esito della scansione, al numero di falsi ed al tempo impiegato.
In questa tesi si vogliono invece analizzare le caratteristiche dell'implementazione, determinando quali di queste hanno influito sui risultati e quali sono risultate superflue. Inoltre si osserva la scarsa adozione dei tool esistenti, che risulta essere in contrasto con quanto esposto fino a quel momento nella tesi e si cerca di fornire delle motivazioni valide per spiegare tale fenomeno.

Viene infine mostrato Vulture, un tool attualmente in sviluppo progettato in collaborazione con EURECOM nell'ambito di una ricerca sul rapporto tra l'analisi statica e le vulnerabilità di tipo Http Parameter Pollution, la cui implementazione è iniziata nel corso di questo lavoro di tesi. Vulture nasce con l'obiettivo di automatizzare il processo di scoperta di potenziali vulnerabilità nel codice sorgente al fine di fornire allo sviluppatore un feedback sufficientemente veloce da consentire l'uso del tool durante lo sviluppo software, garantendo una sorta di continuous integration[6] rivolta all'analisi di sicurezza. Il lavoro relativo a Vulture effettuato nel corso di questa tesi è soprattutto di progettazione, ovvero la scelta delle caratteristiche implementative adatte alla realizzazione del tool in base alle analisi effettuate in precedenza su tool esistenti ed in base alle caratteristiche dell'interprete PHP.

Negli ultimi capitoli della tesi vengono infine discusse le problematiche individuate nell'ambito dell'analisi statica e si cerca di capire lo stato attuale del settore, se è maturo e competitivo oppure se esistono margini per un nuovo business che approcci le esigenze moderne delle software house.

Capitolo 2

Sicurezza di applicazioni web

> If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology.
>
> — Bruce Schneier, 'Applied Cryptography' author

La diffusione globale di internet è un fenomeno in costante crescita, determinato dalle sempre più agevoli condizioni di accesso e coadiuvato dall'interesse per i servizi che la rete offre.
Le applicazioni web sono parte fondamentale di questo processo, la loro evoluzione nel corso degli anni è stata un fattore determinante per la crescita della rete. Servizi sempre più complessi hanno favorito l'interazione con gli utenti e la crescita di nuove opportunità di business. Ciò ha catturato l'interesse di individui con cattive intenzioni, per tale motivo è nata l'esigenza di meccanismi che potessero assicurare la sicurezza dei dati.

Un'applicazione web consiste in un software, posizionato su un server, al quale si accede tramite un'interfaccia web. L'accesso si basa sul protocollo HTTP ed è effettuato da vari clients. Ad ogni interazione tra il server ed il client viene stabilita una comunicazione, che consiste in una richiesta HTTP (solitamente GET e POST) con una serie di parametri che stabiliscono i valori della richiesta.
Il server riconosce i valori che vengono scambiati e di conseguenza costruisce delle risposte coerenti con le richieste, a seconda delle istruzioni definite nell'applicazione, a loro volta create tramite il linguaggio server-side utilizzato. Il codice sorgente dell'applicazione può fare uso dei parametri forniti per effettuare molteplici operazioni, da query sul database a lettura di file, fino a chiamate sul sistema operativo. Tipicamente il linguaggio server-side produce una risposta in formato HTML, che viene poi spedita al client e mostrata nel browser.
I parametri che vengono scambiati tra client e server però non sono sempre prevedibili, gli utenti possono inserire parametri che portano a comportamenti inattesi nell'applicazione. Tali comportamenti possono essere innocui oppure possono essere dannosi per l'applicazione stessa o per gli utenti che la utilizzano. Possono infatti svelare dati sensibi-

li, compromettere l'applicazione o compromettere le altre applicazioni che lavorano sullo stesso server.

Creare sistemi sicuri non è semplice e comporta la soluzione di numerosi e complessi problemi: dallo sviluppo di un'architettura sicura alla creazione di robusti sistemi crittografici fino alla definizione di policy di sicurezza. Nonostante l'esistenza di queste problematiche, grossa parte delle problematiche di sicurezza di un'applicazione sono dovute ad errata implementazione oppure alla negligenza dello sviluppatore.
Nel corso degli anni il problema della sicurezza dei dati ha assunto dimensioni rilevanti tanto che sono state proposte soluzioni per integrare la sicurezza nel processo di sviluppo software.

2.1 Fondamenti di sicurezza

La sicurezza nel software è basata sui principi di *confidentiality*, *integrity* ed *availability*, solitamente definita con l'acronimo *CIA*.

- Confidentiality: indica il divieto di diffusione di informazioni a soggetti o sistemi non autorizzati. E' condizione necessaria (ma non sufficiente) per garantire la privacy.
- Integrity: indica la certezza che un dato non venga modificato in modo imprevisto da chi non ne possiede l'autorizzazione.
- Availability: indica la disponibilità del dato per chi ne ha l'autorizzazione quando richiesto.

Negli ultimi anni tuttavia, con la motivazione di non riuscire a caratterizzare completamente il termine, è stata messa in discussione la definizione di sicurezza; sono stati proposti ulteriori termini che tengono conto anche di altri parametri. Nel 2002 Donn Parker[34] propose un'alternativa composta da sei termini, aggiungendo ai tre classici principi le nozioni di possession, authenticity e utility (la sestupla verrà in seguito nominata *Parkerian Hexad*). Possession indica la proprietà dei diritti di controllo dei dati, authenticity indica la capacità di accertare la validità del dato, utility indica la capacità di un dato di essere utile per un determinato scopo.

2.2 Vulnerabilità nelle applicazioni web

La presenza di vulnerabilità nel software è dovuta a vari motivi: errate decisioni architetturali ed implementative, mancata conoscenza da parte dello sviluppatore delle problematiche di security, negligenza. Queste ultime due motivazioni sono ancora più veritiere nel mondo degli applicativi web: linguaggi come PHP non hanno una curva di apprendimento ripida e consentono anche ai non professionisti di realizzare applicazioni web.
Molti sviluppatori non conoscono o non si rendono conto delle problematiche di security a

cui vanno incontro se vengono inseriti dati malevoli nelle loro applicazioni. E' un problema principalmente di educazione, i vari libri di programmazione difficilmente si soffermano sull'importanza di scrivere codice sicuro. Allo stesso modo il lavoro di sviluppatore non sempre richiede certificazioni in ambito sicurezza per essere praticato.
Un'altra motivazione che esclude la sicurezza dal processo di sviluppo software è costituita da restrizioni economiche, le quali possono incidere sulle tempistiche e quindi diminuire il tempo da dedicare al testing ed al controllo del codice. Solitamente infatti raggiungere una release stabile del progetto è la massima priorità, mentre la sicurezza non lo è.

Il controllo qualità nei progetti software è focalizzato solitamente sull'adesione ai requisiti imposti in fase di progettazione, non sulle implicazioni che può avere una errata implementazione delle specifiche. Solitamente però la presenza di vulnerabilità non indica una violazione delle specifiche imposte dai requisiti.
Tecnicamente, al fine di rendere un software sicuro, tutte le parti di quel software devono essere sicure, non solo le parti sensibili dal punto di vista della sicurezza come l'autenticazione o la gestione dei pagamenti. E' proprio in questo codice che statisticamente si concentrano le vulnerabilità, quelle in cui la sicurezza non è un requisito.
Un esempio di tale eventualità si ha con la procedura di acquisto prodotti su un e-commerce: non è necessario che solo la parte di acquisto tramite carta di credito sia sicura, un attaccante può sfruttare una vulnerabilità in qualunque punto dell'applicazione per accedere ai dati delle carte di credito degli utenti.

Le vulnerabilità più comuni all'interno di un'applicazione web sono chiamate *taint-style vulnerabilities*. Il nome deriva dal fatto che un dato (*tainted data*) entra nel flusso del programma attraverso una sorgente non fidata e viene passato ad una porzione del programma vulnerabile (detta *sensitive sink*) senza essere prima sanitizzato da una opportuna routine (*sanitization routine*). La mancata sanitizzazione del dato comporta la presenza di una vulnerabilità di tipo taint-style.
Il linguaggio di scripting Perl possiede un *taint-mode*, ovvero una modalità che considera ogni input dell'utente come tainted. Con taint-mode attivo, solo input da provenienti da sinks che vengono validati attraverso espressioni regolari vengono considerati validi.
Questa tecnica ha un evidente grosso punto debole: le espressioni regolari sono molto complesse ed è facile per uno sviluppatore commettere un errore che possa invalidare la sanitization routine (Jovanovic et. al.[32]). Per tale motivo è sconsigliato l'uso di sanitization routine basate su espressioni regolari definite in modo personalizzato ma si consiglia di usare i costrutti forniti dal linguaggio. L'assunzione che indica come sicuro ogni valore che viene passato attraverso un'espressione regolare è oltremodo problematica, fornisce un falso senso di sicurezza, per tale motivo non è sensato riprodurre il taint-mode di Perl su altri linguaggi come misura protettiva.

Le vulnerabilità di tipo taint-style, tra cui injections, cross site scripting e cross site request forgery, sono le più comuni ma non sono le uniche ad affliggere le applicazioni web. OWASP (Open Web Application Security Project)[1], un gruppo composto da volontari che produce tools, standard e documentazione open-source gratuita inerente la web security, ha categorizzato le varie vulnerabilità nella sua Top Ten[33], un progetto rilasciato con

[1] http://www.owasp.org

cadenza triennale che raccoglie 10 vulnerabilità tipiche di applicazioni web. Gli obiettivi di OWASP sono i seguenti:

- Diffondere la cultura dello sviluppo di applicativi web sicuri.
- Contribuire alla sensibilizzazione sia dei professionisti che delle aziende verso le problematiche di web security, attraverso la circolazione di idee, articoli, best-practices e tools.
- Promuovere l'uso di metodologie e tecnologie che consentano di migliorare il livello di sicurezza delle applicazioni web.

OWASP Top Ten è una classificazione accettata a livello globale basata ed è basata sul rischio, che fornisce anche le contromisure per mitigare l'eventuale problematica. Di seguito si riportano le vulnerabilità citate dalla OWASP Top Ten 2010, utili in seguito nella trattazione, corredate da un esempio di come è possibile riscontrarle nell'applicazione web (sebbene l'esempio sia didattico e meno complicato rispetto alla vulnerabilità riscontrata negli applicativi professionali).

2.2.1 Injection

Questa categoria di vulnerabilità raccoglie tutte le casistiche in cui dati non fidati vengono inviati ad un interprete come parte di un comando o di una query. Tali dati possono essere eseguiti dall'interprete e possono condurre all'esecuzione di comandi non voluti oppure all'accesso a dati non autorizzati. Fanno parte di questa categoria le vulnerabilità di tipo SQL injection, LDAP injection e OS injection.
E' molto comune trovare questa tipologia di vulnerabilità in codice legacy, ovvero non più supportato, ed è una vulnerabilità che ha un severo impatto sull'applicazione poichè può portare alla corruzione del sistema, ad un *denial of service* oppure alla perdita di dati.
Di seguito si riporta un esempio di tale vulnerabilità:

```
1 $query = "SELECT * FROM accounts WHERE custID ='" . $_GET["id"] ."'";
2 mysql_query($query);
```

L'applicazione esegue la query sul database MySql sottostante, utilizzando come parametro un valore ottenuto direttamente dall'URL. Tale query espone però l'applicazione ad un possibile attacco di tipo SQL Injection. Infatti inserendo nell'URL una stringa come la seguente

```
1 http://example.com/app/accountView?id=' or '1'='1
```

la query viene interpretata in modo diverso, ritornando tutti i record di quella tabella dal database. Nel caso peggiore un attaccante può utilizzare questa vulnerabilità per eseguire query che alterano i dati nel database, riuscendo ad ottenere il completo controllo dell'applicazione.

Per evitare di incorrere in questa tipologia di vulnerabilità è opportuno utilizzare un API per il dialogo con il database, che si occupa di filtrare i parametri in ingresso alle query. Una soluzione alternativa può essere invece quella di effettuare l'escape dei caratteri speciali usando specifiche sintassi per ogni interprete.

2.2.2 Cross site scripting

Vulnerabilità di tipo Cross site scripting (denominate spesso con l'acronimo XSS, da non confondere con CSS di cascading style sheets) si verificano quando un'applicazione riceve dati di input non fidati e li invia ad un browser senza un'appropriata procedura di controllo e validazione.
XSS consente ad un attaccante di eseguire scripts sul browser della vittima, i quali possono effettuare l'hijacking della sessione utente, possono recuperare cookie di sessione, possono redirezionare l'utente su siti web malevoli o possono effettuare defacing del sito web.
Un esempio di cross site scripting può essere il seguente:

```
1 $page += "<input name='creditcard' type="text" value='" + $_GET["CC"]
    + "'>";
```

Supponendo di avere una pagina che mostra a video il numero di carta di credito di un individuo, ottenuto attraverso i parametri in input dall'URL, l'attaccante potrà semplicemente costruire un URL con un valore del parametro *CC* modificato e fare in modo che l'utente visiti tale URL per effettuare l'hijacking della sessione utente, come nell'esempio seguente:

```
1 '><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.
    cgi?foo='+document.cookie</script>'.
```

Esistono tre diverse tipologie di cross site scripting:

- Stored: Il codice viene iniettato nel server in modo permanente, ad esempio in un database. La vittima riceve quindi lo script malevolo ad ogni visita della pagina che mostra a video quel valore.
- Reflected: Il codice malevolo non viene iniettato nel server ma viene inviato alla vittima attraverso mezzi alternativi, come un email contenente un link. Quando l'utente clicca su tale link il codice viene eseguito.
- DOM based: Un payload malevolo viene eseguito come risultato della modifica del DOM[2] del browser dell'utente. La risposta HTTP in questo caso non cambia ma il codice contenuto nella pagina viene eseguito in modo diverso a causa delle modifiche al DOM.

[2]Document Object Model

E' diverso da stored e reflected XSS poichè in questo caso il payload malevolo non è nella pagina di risposta del server ad una richiesta.

Vulnerabilità di tipo XSS hanno impatto significativo sull'utente, meno significativo sull'applicazione (ad eccezione del caso stored, in cui l'applicazione è direttamente coinvolta). Prevenire vulnerabilità di tipo XSS comporta la separazione tra dati non fidati ed il contenuto che viene inviato al browser. E' quindi necessario effettuare l'escape dei contenuti in input basati su codice HTML o Javascript, a seconda del contesto in cui tali dati verranno poi utilizzati.

2.2.3 Broken authentication and session management

Funzionalità come l'autenticazione e la gestione delle sessioni utente sono spesso implementate non correttamente, consentendo all'attaccante di ottenere passwords, chiavi, tokens di sessione o di impersonificare altri utenti.
Il classico esempio di questa vulnerabilità si verifica quando il timeout della sessione utente non è settato correttamente. Supponendo che l'utente sia loggato nell'applicazione attraverso un browser su un computer e al termine dell'uso si dimentichi di cliccare su logout, un attaccante potrebbe collegarsi al sito attraverso lo stesso computer e ritrovarsi già loggato nell'applicazione, con il profilo del vecchio utente.
Questa tipologia di vulnerabilità ha radici architetturali oltre che implementative, per tale motivo le contromisure consistono nel seguire raccomandazioni e specifiche per la gestione delle sessioni e dell'autenticazione utente.

2.2.4 Insecure direct object references

Una direct object reference si verifica quando uno sviluppatore espone un collegamento ad un oggetto interno, come un file, una directory, una chiave per accedere al database, ecc. Senza le opportune protezioni gli attaccanti possono manipolare tali collegamenti per accedere a dati in modo non autorizzato.
Un esempio di tale vulnerabilità può verificarsi nelle applicazioni di home banking, dove il numero di conto corrisponde alla chiave primaria nel database. Anche se gli sviluppatori hanno utilizzato query SQL che evitano injections, se non esistono controlli aggiuntivi per verificare che l'utente sia il proprietario dell'account e sia autorizzato a visualizzare determinati contenuti, un attaccante potrebbe sfruttare il numero di conto per visualizzare dati provenienti da conti altrui.
Le contromisure consistono nella verifica continua che l'accesso ai dati esposti avvenga effettivamente dagli utenti autorizzati e nell'uso di referenze ad oggetti interni indirette, quali ad esempio indici non collegati a dati sensibili.

2.2.5 Cross site request forgery

Un attacco di tipo cross site request forgery (CSRF) forza una vittima loggata nell'applicazione web ad eseguire richieste HTTP costruite dall'attaccante per uno scopo ben preciso (tramite tags di tipo immagine, XSS o altre tecniche), generalmente ad insaputa della vittima. L'applicazione a questo punto reagisce interpretando la richiesta come legittima, e la richiesta può accedere ad ogni dato a cui la vittima può accedere.
Si riporta un esempio di tale problematica:

```
http://example.com/app/transferFunds?amount=1500&destinationAccount
    =4673243243
```

L'applicazione di home banking esegue un trasferimento fondi da un account ad un altro mediante l'uso dei parametri riportati. In questo caso non vi è nulla di segreto nella richiesta. Se l'attaccante fosse in grado di lanciare sul computer della vittima loggata al sito una richiesta di questo tipo, con il proprio numero di conto nel campo destinationAccount, potrebbe trasferire la cifra riportata a se stesso. Per fare ciò, potrebbe forgiare un tag di tipo immagine apposito e fare in modo che la vittima lo visualizzi da loggata.
L'esempio sottostante mostra un tag immagine adatto allo scopo:

```
<img src="http://example.com/app/transferFunds?amount=1500&
    destinationAccount=attackersAcct#" width="0" height="0" />
```

Prevenire CSRF comporta la creazione di un token non prevedibile nel corpo o nell'URL di ogni richiesta HTTP. Tale token dovrebbe essere univoco per sessione utente, ma è ancora meglio se è univoco per ogni richiesta. Con la presenza di tale valore non è più possibile per l'attaccante la creazione di una richiesta valida da fare eseguire di nascosto alla vittima.

2.2.6 Altre vulnerabilità

Le altre vulnerabilità citate in OWASP Top Ten ma non riportate nella sezioni precedenti sono le seguenti:

- *Security misconfiguration*: Le applicazioni web si basano su uno stack, ovvero un insieme di programmi che lavorano a diversi livelli. Molti di essi devono essere configurati correttamente, poiché non tutti vengono forniti di default con un setup sicuro. Inoltre devono essere mantenuti ed aggiornati. Questa tipologia di vulnerabilità include tutti i casi in cui tale software non viene configurato o mantenuto correttamente.
- *Insecure cryptographic storage*: Molte applicazioni web non proteggono attraverso l'uso di cifratura i dati sensibili, come numeri di carte di credito o credenziali di autenticazione. Gli attaccanti possono quindi rubare o modificare tali dati per eseguire furti di identità, frodi ed altri crimini.

- *Failure to restrict URL access*: E' comune per un'applicazione web controllare i permessi di accesso all'URL prima di visualizzare contenuti protetti. Tuttavia le applicazioni necessitano che questi controlli vengano eseguiti ogni volta che queste pagine vengono visualizzate o gli attaccanti saranno in grado di costruire URL appositi per accedere a tali risorse senza i permessi.
- *Insufficient transport layer protection*: Le applicazioni spesso falliscono nel proteggere la confidenzialità e l'integrità del traffico di rete. Questo poiché talvolta utilizzano algoritmi non corretti, certificati non validi o scaduti, non implementano SSL oppure falliscono nell'implementare le best practice per questa problematica.
- *Unvalidated redirects and forwards*: Può capitare che un'applicazione redirezioni l'utente verso altre pagine o siti web, usando dati non sicuri per determinare le pagine di destinazione. Senza l'appropriata validazione un attaccante può redirezionare la vittima verso phishing verso siti contenenti malware.

2.2.7 Http Parameter Pollution

Http Parameter Pollution (HPP) è una vulnerabilità presentata per la prima volta nel 2009 da Stefano di Paola e Luca Carrettoni[30] ad OWASP AppSec Europe 2009. Sebbene troppo recente per essere inclusa nella OWASP Top Ten, questa vulnerabilità è la motivazione che ha portato alla nascita di Vulture, il tool che verrà presentato successivamente in questa tesi.
HPP, descritta in modo esaustivo da Balduzzi[23], consiste nell'inserimento nell'input di un'applicazione web una particolare query string a parametri encodati, la quale, se l'applicazione non esegue una sanitizzazione dei valori in input, può consentire ad un utente malevolo di comprometterne la logica.

La RFC 3986[25] specifica che la query string è la parte dell'URI compresa tra il carattere ? e la fine dell'URI stesso. Normalmente la query string è composta da una serie di coppie chiave=valore che identificano i valori in input, separate da un carattere & oppure ;. Al fine di evitare incomprensioni con i separatori, ogni carattere speciale viene encodato in forma esadecimale.
A seconda della tecnologia in cui l'applicazione è stata sviluppata, la query string viene però interpretata in modo differente: se esistono coppie chiave=valore con la stessa chiave, l'applicazione può prendere in input il primo dei due valori, l'ultimo, oppure una combinazione di entrambi.
Nella tabella 2.1 si mostra in dettaglio il funzionamento dell'applicazione a seconda della tecnologia usata.

La presenza di più valori per la stessa chiave della query string può portare a comportamenti indesiderati dell'applicazione, è quindi fonte di attacchi da parte di utenti malintenzionati.
Uno scenario tipico consiste nel far si che la vittima dell'attacco venga redirezionata verso un URL che sfrutta una vulnerabilità di tipo HPP. Supponendo che l'applicazione in questione sia un sistema scritto in JSP per individuare il giorno migliore in cui effettua-

Tecnologia e server	Metodo	Precedenza
ASP/IIS	Request.QueryString(par)	Tutti (separati da virgole)
PHP/Apache	$_GET[par]	Ultimo valore
JSP/Tomcat	Request.getParameter(par)	Primo valore
Perl(CGI)/Apache	Param(par)	Primo valore
Python/Apache	getValue(par)	Tutti (in una lista)

Tabella 2.1: HPP: Precedenza in caso di parametri con lo stesso nome

re un meeting, che riceve in input un singolo parametro chiamato *meeting_id*, al fine di identificare univocamente il meeting stesso, l'attaccante potrebbe effettuare HPP sulla query string con lo scopo di modificare i valori dei link che segnalano il giorno di preferenza dell'utente. Nel normale funzionamento dell'applicazione i link generati sono i seguenti:

```
1 http://example.com/choice.jsp?meeting_id=1234
2
3 Link 1: <a href="http://example.com/choice.jsp?meeting_id=1234&day
    =20111201">1 dicembre 2011</a>
4 Link 1: <a href="http://example.com/choice.jsp?meeting_id=1234&day
    =20111202">2 dicembre 2011</a>
```

L'attaccante, aggiungendo alla query string il valore encodato *day=20111202*, rende la scelta dell'utente indifferente grazie all'attacco di tipo HPP:

```
1 http://example.com/choice.jsp?meeting_id=1234%26day%3D20111202
2
3 Link 1: <a href="http://example.com/choice.jsp?meeting_id=1234\%26day
    \%3D20111202&day=20111201">1 dicembre 2011</a>
4 Link 1: <a href="http://example.com/choice.jsp?meeting_id=1234\%26day
    \%3D20111202&day=20111202">2 dicembre 2011</a>
```

Nel secondo caso, visto che il funzionamento di JSP con i doppi parametri in query string consiste nel scegliere sempre il primo valore tra quelli con la stessa chiave, qualunque link l'utente clicchi la sua scelta sarà sempre salvata come 1 dicembre 2011.
Esistono alcune varianti di HPP che si rifanno però sempre all'injection di valori multipli nella query string ed alle precedenze nell'uso di tali valori da parte dell'applicazione. E' quindi compito dello sviluppatore controllare la reazione dell'applicazione all'invio di parametri multipli.

2.3 Security development life cycle

Tutte le realtà aziendali che si occupano di software implementano un modello che definisce le fasi entro cui lo sviluppo si verifica. Tali fasi sono definite nel software development life cycle (SDLC).
Esistono diversi modelli di software development life cycle:

- A cascata
- A spirale
- Iterativo ed incrementale
- Agile
- Code and fix

Tutti questi modelli però non contemplano la fase di analisi della sicurezza dell'applicazione, bensì relegano tale fase al termine del processo di sviluppo. Ciò è inefficiente poiché influisce sui costi (una modifica a prodotto completato è sicuramente più costosa di una modifica eseguita durante lo sviluppo) e comporta una perdita di controllo (modificare il prodotto in uno stage così avanzato del processo comporta dover eseguire nuovamente tutti i test). L'immagine 2.1 mostra le problematiche del modello patch-and-penetrate.

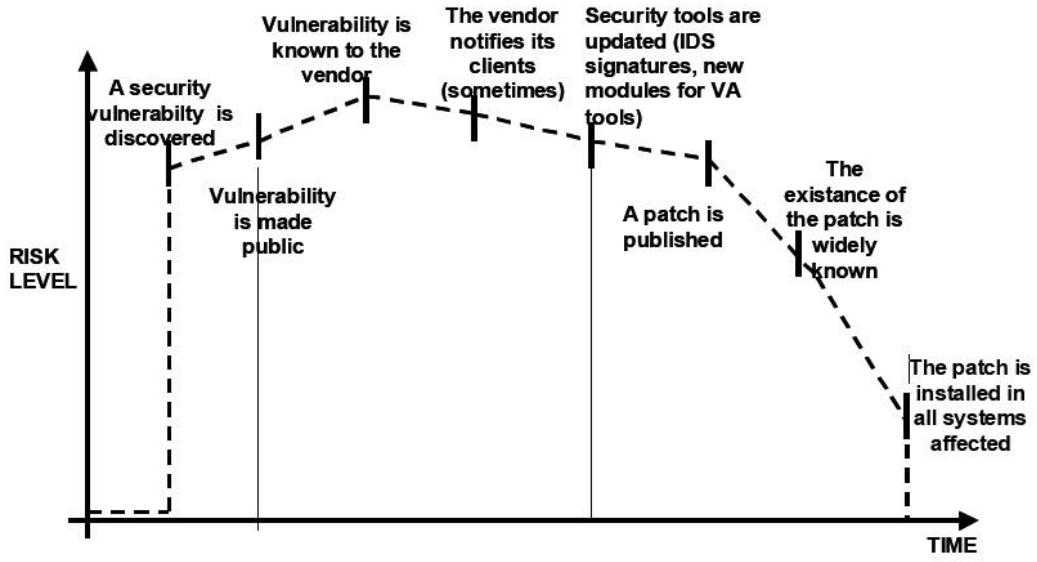

Figura 2.1: Modello patch-and-penetrate

Per tali motivi è opportuno che l'analisi di sicurezza venga implementata nello sviluppo, introducendo un diverso software development life cycle.

L'immagine 2.2 mostra come la sicurezza non sia una fase all'interno del processo di sviluppo, bensì un processo, da attuare costantemente.
E' fondamentale identificare il prima possibile un problema di sicurezza, poiché ciò comporta costi minori di fixing e riduce in modo drastico la finestra di esposizione, presente invece nel modello patch-and-penetrate, dominante fino a pochi anni fa.
Introdurre l'analisi di sicurezza nel SDLC non è però semplice, occorre una maggiore conoscenza delle problematiche e qualche sistema in grado di automatizzare la ricerca di errori comuni, che abbia come requisito la rapidità di scansione e che possa fornire feedbacks immediati agli sviluppatori.

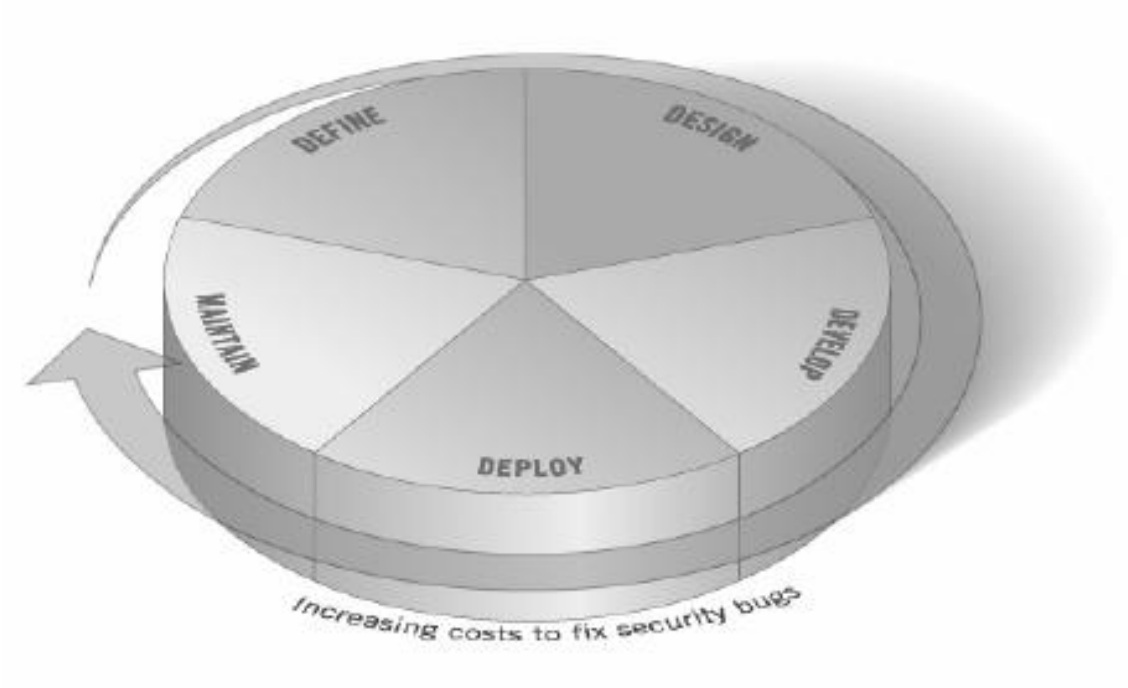

Figura 2.2: Software development life cycle

Per questo compito, oltre alla sempre necessaria formazione degli sviluppatori, l'analisi statica può rivelarsi un aiuto efficace.

Capitolo 3

Analisi Statica

> As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.
>
> — Maurice Wilkes, Inventore di EDSAC, 1949

Tutti i progetti software condividono una caratteristica fondamentale: possiedono un codice sorgente che ne definisce il funzionamento. Il codice sorgente è costituito da una serie di istruzioni scritte in un linguaggio di programmazione che vengono interpretate da un compilatore e successivamente eseguite. Il codice sorgente di un software risiede tipicamente su uno o più files di testo.
Il codice sorgente non è esente da errori bensì ha l'intrinseca proprietà di possedere difetti. Sin dagli albori della programmazione software gli sviluppatori hanno avuto a che fare con tali difetti, individuando un rapporto di proporzionalità diretta tra il numero di questi ultimi ed il numero di righe di codice scritte per un determinato software. L'aumentare della complessità dei programmi e la necessità di affidabilità hanno reso il controllo dei difetti fondamentale nell'industria del software, tanto che è opportuno che prima di un rilascio determinati standard di qualità siano rispettati.
Al fine di ridurre il quantitativo di difetti nel software e di aderire agli standard i programmatori hanno pensato di sviluppare altro software in grado di analizzare il codice sorgente di un programma durante la fase di sviluppo.
Tale analisi è detta *analisi statica* ed è una tecnica che consiste nell'ispezionare automaticamente il codice sorgente di un software senza però eseguirlo. Il grosso vantaggio di tale tecnica consiste nella sua applicazione alla radice del processo di sviluppo, in contrasto alle esistenti tecniche di testing che vedevano posticipata la correzione dei difetti alla fase di pre-rilascio. Anticipare l'identificazione del difetto software comporta minori costi di correzione; è proprio questo il motivo del successo dell'analisi statica.

3.1 Il problema

Alan Touring[36] nel 1937 affermò che esistono due tipologie di problemi: problemi decidibili, per i quali esiste un algoritmo (quindi una procedura eseguibile con un numero finito di passi) in grado di risolverli e problemi non decidibili, per i quali non esiste alcun algoritmo in grado di dare una risposta in tempo finito su tutte le istanze del problema. Henry Gordon Rice estese tale teorema[35], asserendo che ogni domanda non banale riguardante il comportamento di un programma è indecidibile. Di conseguenza non esiste alcun metodo meccanico per determinare se un dato programma può o non può generare errori di esecuzione.
L'analisi statica è quindi un problema non decidibile. Alcuni fattori non possono infatti essere valutati se non dinamicamente e dipendono strettamente dalle condizioni di input. Essendo non decidibile, l'analisi statica genererà sempre falsi positivi e negativi e non garantisce che i risultati siano corretti. L'accuratezza consiste nel determinare un buon livello di approssimazione al fine di ridurre i falsi.

3.2 Storia

Il primo approccio all'analisi statica è storicamente da attribuire al tool di Unix *grep*[1], che effettua una ricerca in uno o più file di testo di linee corrispondenti ad uno o più modelli specificati con espressioni regolari o stringhe letterali. Grep non conosce nessun dettaglio dei files che sta esaminando, li interpreta semplicemente come files di testo, di conseguenza sebbene possa produrre risultati apprezzabili con determinati pattern di ricerca non si può considerare come un vero e proprio tool di analisi statica.
Per aumentare la fedeltà nei risultati è importante che vengano prese in considerazione le regole lessicali specifiche del linguaggio, al fine di poter distinguere tra i vari costrutti.
La vera nascita delle tecniche di analisi statica viene solitamente attribuita al tool Lint, sviluppato da Stephen C. Johnson e rilasciato alla fine degli anni '70. Lint fu realizzato allo scopo di segnalare come sospetti alcuni costrutti nel sorgente in linguaggio C, come la mancanza di punti e virgola, parentesi, cast impliciti, ecc. Lint era integrato con il processo di compilazione, soluzione che sembrava essere la migliore per riportare segnalazioni relative al codice e che ne contribuì alla diffusione.
Purtroppo le limitate capacità di analisi, quali ad esempio l'obbligo di eseguire la scansione un file per volta, fecero si che Lint riportasse un elevata percentuale di rumore tra i risultati, ovvero valori corretti dal punto di vista dell'analisi ma irrilevanti per lo sviluppatore al fine di correggere difetti. Ciò si tradusse nella necessità di eseguire dei controlli manuali sui risultati di Lint, esattamente la situazione che Lint si era proposto di eliminare. Per tale motivo Lint non fu mai adottato globalmente come tool per l'individuazione di difetti.
Nei primi anni 2000 una seconda generazione di tools emerse, che si è evoluta fino ad oggi. Gli sviluppatori intuirono che era necessario comprendere attraverso il software di analisi maggiori dettagli relativi al funzionamento del programma. Produssero tools in grado di

[1] General Regular Expression Print

analizzare più files contemporaneamente e di identificare i percorsi di flusso dei dati, ma si scontrarono con la problematica che da sempre caratterizza l'analisi statica: il necessario compromesso da attuare tra performance ed accuratezza. L'efficacia delle tecniche di analisi statica è altamente condizionata dal fatto che devono essere gli sviluppatori ad utilizzarle, poichè prima si è in grado di identificare il difetto e minore sarà il costo della sua correzione.

3.3 Applicazioni

Sebbene le tecniche di analisi statica nacquero allo scopo di individuare difetti e di aderire a standard nella stesura del codice, molteplici successivi utilizzi vennero identificati ed implementati.
Attualmente l'analisi statica è utilizzata negli IDE[2] per evidenziare la sintassi e le parole chiave, restituendo al programmatore una migliore visualizzazione del sorgente del programma ed aiutandolo a rintracciare facilmente errori di battitura.
Sempre negli IDE viene utilizzata per segnalare eventuali errori nell'adesione alle regole definite da standard di scrittura del codice, come le spaziature, i rientri, la lunghezza delle righe ed il posizionamento delle parentesi. Sebbene questi standard possano risultare ai più superflui, quando il progetto in sviluppo viene realizzato da un numero elevato di sviluppatori è importante uniformare il codice al fine di ottenere una codebase leggibile.
I software di ottimizzazione del codice sfruttano l'analisi statica per determinare porzioni di codice che possono essere eseguite più velocemente se riorganizzate; tale analisi viene svolta valutando come le istruzioni riempirebbero la pipeline della CPU e riordinandole di conseguenza.
L'analisi statica è utilizzata anche a fini di generazione della documentazione; tali librerie leggono il contenuto delle annotations e dei commenti all'interno del codice sorgente per generare documenti che descrivono la struttura del programma. Tra questi si ricorda Doxygen[3], una delle librerie più utilizzate a questo scopo.
L'analisi statica è infine anche usata dai programmatori per capire il funzionamento, per calcolare le metriche e per rifattorizzare il codice sorgente.
In caso di software che possiede requisiti di sicurezza, ed ultimamente sempre più spesso visto il diffondersi di applicazioni critiche sotto questo punto di vista, l'analisi statica consente di individuare codice potenzialmente vulnerabile.

A seconda dell'utilizzo e dell'implementazione un'analisi statica varia totalmente per velocità di esecuzione, dalla rapidità di un'esecuzione per il controllo della sintassi alla lentezza di un'esecuzione alla ricerca di problematiche di sicurezza che considera il flusso dei dati. La complessità del task risulta essere solitamente direttamente proporzionale al tempo necessario per l'esecuzione e ciò influenza gli utilizzi di tale tipologia di analisi.

[2]Integrated Development Environment
[3]http://www.doxygen.org

3.4 Analisi statica vs. analisi dinamica vs. code review

Le tecniche di analisi statica analizzano il sorgente di un programma in modo automatico senza eseguirlo. Prima della nascita di tali tecniche gli sviluppatori effettuavano un controllo manuale sul codice chiamato *code review*.

La code review può essere eseguita da più sviluppatori (*peer review*) oppure da un solo sviluppatore. E' una procedura complessa, che ha come requisito fondamentale la piena conoscenza delle decisioni architetturali prese durante la progettazione e la scrittura del codice oltre ad un'ottima padronanza del linguaggio in analisi.
Esistono due categorie di code review: *formal code review* e *lightweight code review*. La prima categoria richiede un dettagliato processo di analisi suddiviso in molteplici fasi. Tale metodologia comporta l'analisi di copie stampate del materiale ed è svolta da più partecipanti che contemporaneamente analizzano il codice.
La seconda categoria richiede meno formalismi rispetto alla precedente e viene svolta solitamente durante il normale processo di sviluppo. All'interno di essa si trovano le seguenti pratiche:

- Over-the-shoulder: uno sviluppatore osserva il codice che l'altro sta scrivendo per segnalare eventuali problemi
- Email pass around: un SCM[4] invia tramite email il nuovo codice inserito nella codebase ad un soggetto che si occupa di effettuare la review.
- Pair programming: Due sviluppatori scrivono codice contemporaneamente sulla stessa workstation.
- Tool-assisted code review: Sviluppatori e reviewers usano tools in grado di effettuare code review collaborativa.

La problematica di questa tipologia di analisi consiste nel tempo che richiede; i dati raccolti dai maggiori operatori del settore stimano che in media si possa effettuare code review su 150 linee di codice per ogni ora, fino a rimuovere l'85% dei difetti presenti nel software.

L'analisi dinamica è una tecnica che consiste nell'osservare il comportamento del software durante la sua esecuzione. Al fine di rendere tale tecnica effettiva è necessario che il programma venga eseguito con diversi input. L'analisi dinamica è una tecnica precisa, che non comporta approssimazione poichè osserva l'esatto comportamento runtime dell'applicazione. Lo svantaggio dell'analisi dinamica è la sua specificità: i risultati proposti riguardano solo ed esclusivamente quell'esecuzione, non c'è garanzia che la test suite utilizzata esegua effettivamente tutti i possibili data flow all'interno del software e che quindi esegua ogni porzione del codice in esame. L'approccio definito dall'analisi dinamica è particolarmente adatto per il testing ed il debugging.

[4]Source Code Management System

Analisi statica ed analisi dinamica sono due approcci complementari che possono essere applicati allo stesso problema, i risultati hanno però diverse proprietà e l'esecuzione di ognuno ha diversi costi. Per tale ragione esistono soluzioni in grado di combinare analisi statica ed analisi dinamica al fine di ridurre i difetti tipici delle due tecniche e fornire risultati più attendibili.
Solitamente la combinazione di queste due tipologie consiste in tool di analisi dinamica che ispezionano dato in ingresso ed in uscita, usando la conoscenza ottenuta dall'analisi statica per aumentare l'accuratezza.
Un esempio di questa tipologie di tool è Saner, sviluppato da Balzarotti et. al.[24], il quale utilizza un tool di analisi statica per effettuare il riconoscimento di routine custom di sanitizzazione del codice, ed un'analisi dinamica successiva per valutarne l'efficacia.

Capitolo 4

Applicazione dell'analisi statica alla sicurezza di applicazioni web

> If you spend more on coffee than on IT security, you will be hacked. What's more, you deserve to be hacked.
>
> — Richard Clarke, White House Cybersecurity Advisor

Analizzando i reports di più attacchi eseguiti con successo alla sicurezza di applicazioni web si è notato che molti di questi non sono rivolti alla scoperta di nuove tipologie di vulnerabilità, bensì alla ricerca di vulnerabilità note.
Sebbene sia facile imputare la continua presenza di tali vulnerabilità alla negligenza dello sviluppatore, il vero problema è dato dal fatto che le tecniche per evitare queste problematiche non sono codificate nel processo di sviluppo software e non è pensabile fare affidamento sulla memoria dello sviluppatore per evitarle. E' questo il motivo per cui occorrono dei sistemi in grado di rilevare tali vulnerabilità direttamente nel processo di sviluppo, ed è in questo caso che l'analisi statica entra in gioco.

I primi approcci all'analisi statica orientata alla sicurezza si possono trovare nei tools RATS[31], ITS4[27] e Flawfinder[17], capaci di effettuare il parsing del codice sorgente al fine di trovare chiamate a funzioni pericolose. Dopo aver trasformato in tokens il codice sorgente (il primo step che anche il compilatore esegue), questi tools effettuavano una comparazione tra lo stream di tokens generato ed una libreria di costrutti vulnerabili. Sebbene questi approcci basati sull'analisi lessicale furono un passo avanti rispetto a grep, producevano un elevato numero di falsi positivi. Uno stream di tokens è certamente meglio di uno stream di caratteri, ma nemmeno questi tools avevano una minima conoscenza del funzionamento del programma.
Con il passare degli anni i tool per l'analisi statica orientata alla sicurezza sono cresciuti in complessità e sono diventati più sofisticati. La vera evoluzione si è avuta con la presa in considerazione del contesto delle chiamate e delle informazioni semantiche di un programma. Questo ha reso possibile l'individuazione delle condizioni entro cui una vulnerabilità si può manifestare, aumentando l'accuratezza e l'efficienza.

Costruendo un AST[1] dal codice sorgente questi tool possono effettivamente considerare la semantica del programma in esame. Dall'AST è possibile eseguire diverse tipologie di analisi:

- Local analysis: esamina il programma una funzione per volta e non considera le relazioni tra le funzioni.
- Module-level analysis: considera un modulo/classe per volta ma non considera le chiamate tra i moduli.
- Global-level analysis: considera il programma per intero prendendo in considerazione tutte le chiamate tra funzioni.

A seconda della tipologia di analisi dipende la quantità di contesto che il tool deve considerare, più contesto significa meno falsi positivi ma anche più computazione.
Un buon tool per l'analisi statica orientata alla sicurezza deve essere facile da usare ed i risultati devono essere comprensibili agli sviluppatori che non conoscono approfonditamente le problematiche di sicurezza. Per far si che un tool di analisi statica venga utilizzato come parte attiva nello sviluppo è importante che il tempo di computazione non sia troppo elevato viceversa rischia di essere utilizzato saltuariamente, fallendo il l'obiettivo per il quale è stato ideato.

4.1 Tecniche di analisi

In letteratura sono presenti diversi approcci per l'esecuzione di analisi statica finalizzata alla scoperta di vulnerabilità di sicurezza. Tutti sono però contraddistinti dalla seguenti tre fasi:

- Costruzione del modello
- Analisi
- Report dei risultati

4.1.1 Costruzione del modello

La fase di costruzione del modello riguarda l'astrazione che è necessario effettuare sul codice sorgente. Come esposto in precedenza esistono tool che lavorano direttamente sul codice, il più primitivo dei quali è grep, tuttavia lavorare direttamente su di esso è problematico poiché è necessario costruire espressioni regolari complesse e non si ha una visione d'insieme dell'intero programma e delle procedure.
Nell'analisi statica solitamente si pre-processa il codice per ottenere una rappresentazione ad un livello più basso, sfruttando le normali operazioni eseguite dal compilatore:

[1] Abstract Syntax Tree

1. Line reconstruction: nei linguaggi che consentono spaziature arbitrarie tra gli identificativi è necessaria la presenza di una fase che ricostruisca le linee restituendo una forma canonica. In alcuni vecchi linguaggi questa fase eseguiva anche una normalizzazione nel caso venisse usato lo stropping[2].

2. Analisi lessicale: il codice viene pulito da elementi inutili come spaziature e commenti e trasformato in uno stream di tokens. Ogni token è una singola unità atomica del linguaggio, ad esempio una parola chiave, un identificativo oppure il nome di un simbolo. Tool come RATS, ITS4 e Flawfinder si limitavano ad utilizzare il risultato di questa fase come input per la propria analisi.

3. Analisi sintattica: lo stream di tokens viene trasformato in un *parse tree*, ovvero una rappresentazione ad albero del codice sorgente. Viene così determinata la struttura dello stream di tokens in input grazie all'uso di una data grammatica formale. L'elemento che esegue questa operazione è detto *parser*.

4. AST: il parse tree creato nella fase precedente viene trasformato in un *abstract syntax tree*. Un AST è un albero simile al parse tree ma pulito di tutti i tokens non utili per l'analisi semantica.

5. Analisi semantica: Ad ogni token viene attribuito un significato al fine di creare una tabella dei simboli. Questa fase esegue il controllo dei tipi di dato (type checking), l'associazione tra le referenze delle funzioni e le loro definizioni (object binding), il controllo che tutte le variabili siano state inizializzate prima dell'uso (definite assignment).

6. Analisi del flusso di controllo: Tutti i possibili percorsi che possono essere eseguiti all'interno del codice vengono tradotti in una serie di *Control flow graphs*. Il flusso di controllo tra le funzioni è raccolto nei *call graphs*.

7. Analisi del flusso dei dati: L'analisi controlla come i dati si muovono all'interno del programma. Vengono a tale scopo usati i grafici realizzati nella fase precedente. Il compilatore usa tale analisi per allocare i registri, rimuovere codice non utilizzato ed ottimizzare l'uso di processore e memoria.

Nonostante il compilatore esegua tutte queste fasi per l'interpretazione del codice sorgente, i tool di analisi statica generalmente si limitano ad alcune di queste operazioni prima di effettuare l'analisi. In base al dato che si analizza esistono innumerevoli approcci a tale fase, ognuno con le proprie caratteristiche e di conseguenza con risultati molto differenti tra loro.

4.1.2 Analisi

Esistono generalmente due approcci per realizzare l'analisi statica di un'applicazione: *local analysis* o *intraprocedural analysis* e *global analysis* o *interprocedural analysis*.

[2]Tecnica che consiste nell'utilizzare una stringa sia come parola chiave che come identificatore.

La local analysis si occupa di analizzare una funzione individualmente, tenendo traccia delle proprietà dei dati nella funzione e delle condizioni per le quali una funzione può essere chiamata. Tracciare tali proprietà diventa però problematico in caso di loop e branches, siccome è necessario salvare lo stato di tutti i dati in ogni momento dell'esecuzione la quantità degli stessi diventa ingente rendendo l'approccio poco praticabile. Riducendo la precisione tuttavia si possono ottenere comunque risultati apprezzabili con quantità di dati inferiori, astraendo le proprietà del programma che non sono di interesse (abstract interpretation) come può essere l'ordine in cui le istruzioni vengono eseguite (flow-insensitive analysis).

A seconda dello stato globale del programma una funzione può riportare risultati differenti. La global analysis consiste nel comprendere il contesto entro il quale una funzione viene eseguita, per valutare le implicazioni conseguenti. A tale scopo è possibile eseguire una *whole-program analysis* che consiste nel combinare in un'unica funzione tutto il codice che viene eseguito sostituendo le chiamate a funzioni con i relativi metodi. Tale tecnica non viene spesso utilizzata poiché porta alla generazione di un unico blocco di codice sorgente molto lungo che richiede molta memoria e molto tempo per essere analizzato.
Una tecnica alternativa è chiamata *function summaries* e consiste nel trasformare una funzione in un insieme di precondizioni e postcondizioni, usando la conoscenza ottenuta tramite la local analysis (talvolta infatti le due tecniche vengono combinate per affinare i risultati). Quando viene eseguita l'analisi si tengono in considerazione solo tali elementi per determinare gli effetti che la funzione ha sull'intero programma.

Dopo aver determinato un'approccio all'analisi statica occorre definire delle policy affinché sia possibile rintracciare all'interno del flusso di istruzioni le eventuali vulnerabilità. Ogni vulnerabilità viene quindi analizzata singolarmente e vengono individuati gruppi di regole che ne permettono l'individuazione.
Pixy[32], un tool di analisi statica orientato alla scoperta di vulnerabilità, richiede che le regole vengano specificate attraverso un file di testo, che definisce le funzioni di sanitizzazione ed i sinks. Altri tool però lavorano in modo differente, basandosi sulle annotations (Splint[20] ad esempio), altri ancora accettano la definizione delle regole mediante l'uso di un linguaggio apposito (solitamente PQL[19]).

4.1.3 Report dei risultati

Al termine della fase di analisi è necessario che i risultati vengano processati per essere mostrati all'utente. Questa fase è molto importante, il valore di un software di analisi statica è infatti direttamente collegato alla sua capacità di riportare risultati in modo comprensibile ed immediato per l'utente.
Tra i possibili risultati di un'analisi figurano: falsi positivi, falsi negativi, veri positivi e veri negativi. I falsi positivi si verificano quando una parte del codice è segnalata impropriamente come vulnerabile, i falsi negativi quando non viene rilevata la presenza di una vulnerabilità esistente. La seconda categoria di falsi è maggiormente problematica poiché fornisce un falso senso di sicurezza. I veri positivi indicano una vulnerabilità

correttamente identificata, i veri negativi indicano l'assenza di una segnalazione per una porzione di codice sicura.

In vari tool i risultati vengono categorizzati in base alla criticità, determinata tenendo conto dell'accuratezza con cui tale vulnerabilità viene rilevata, in altri si tiene conto della profondità, ovvero del numero di chiamate e di loop/branches da analizzare.
Talvolta insieme alla segnalazione viene mostrato un help contestuale che motiva all'utente la presenza della stessa, al fine di renderne maggiormente comprensibile la natura.

Capitolo 5

Analisi statica di codice PHP

> I was really, really bad at writing parsers. I still am really bad at writing parsers.
>
> — Rasmus Lerdorf, PHP author

5.1 PHP

L'8 giugno del 1995 con un messaggio su Usenet Rasmus Lerdorf annunciava la disponibilità di Personal Home Page Tools versione 1.0 (PHP Tools 1.0), la prima release ufficiale di PHP[22]. Questo set di files scritti in C permetteva a Lerdorf di registrare le visite al proprio Curriculum Vitae senza per forza dover accedere alle statistiche del server.
Lerdorf decise di rilasciare PHP Tools sotto licenza GPL, allo scopo di organizzare un gruppo di utenti in grado di fare crescere la propria creazione. Pochi mesi dopo l'annuncio di Personal Home Page Tools, Lerdorf annunciò il rilascio di un parser di nome FI (form-interpreter) da lui stesso sviluppato allo scopo di far interagire le pagine web con mSQL (predecessore dell'attuale mySQL). PHP prese a quel punto il nome PHP/FI, ispirandosi all'acronimo TCP/IP.

L'idea di integrare mSQL all'interno delle pagine web fu indubbiamente ciò che contribuì alla rapida diffusione di PHP e permise la creazione di quel gruppo di sviluppatori che da alcuni mesi Lerdorf cercava di ampliare. Con l'avvento della versione 2.0 il set di script PHP e il parser FI vennero completamente riscritti ed il progetto iniziò a diffondersi globalmente conquistando il traguardo della presenza sull'1% dei domini web. I file di PHP 2.0 avevano estensione .phtml ed il parser FI poteva comunicare con più di una tipologia di database (mSQL, Postgres95 e il neonato mySQL).
La vera svolta nel progetto però avvenne alla fine del 1997 quando due israeliani (Zeev Suraski e Andi Gutmans) svilupparono Zend Engine, un nuovo parser che nel giro di 8 mesi sostituì il parser di PHP/FI 2.0. Nel 1993 venne rilasciato PHP 3, che oltre a segnare l'esplosione di PHP come linguaggio di scripting per il web, segnò la fine dell' era Lerdorf all'interno del team di sviluppo. Infatti il creatore di PHP iniziò a defilarsi mentre nel

team crescevano le personalità di Suraski e Gutmans. PHP 3 fu anche la release che cambiò il significato dell'acronimo PHP, non più Personal Home Page ma, PHP: Hypertext Preprocessor.
Nel 2000 venne rilasciata la quarta versione di PHP, con notevoli miglioramenti sotto il fronte delle API e della velocità di esecuzione. Un grosso cambiamento della versione 4 riguardò la licenza, GNU General Public License (GPL) venne sostituita da PHP4 License, maggiormente restrittiva sebbene sempre open source. Fu questa release a consolidare il ruolo di PHP nel mondo dei linguaggi di programmazione orientati al web. Quattro anni dopo fu rilasciato PHP 5, con un migliorato supporto alla programmazione ad oggetti e il nuovo supporto ai web services.
La versione attuale di PHP è la 5.3. In questa release il supporto agli oggetti è stato esteso con l'aggiunta dei namespace, ovvero un sistema che permette di raggruppare variabili, classi e funzioni all'interno di un determinato spazio dei nomi al fine da diminuire le possibilità di collisione. Inoltre risultano ora supportati i late static binding e le closures.
Nonostante questi miglioramenti il linguaggio vive di un contrasto perenne tra le community di sviluppatori. Ritenuto spesso male organizzato e poco evoluto rispetto ai rivali che via via si stanno affermando in ambito web come Python e Ruby, viene molto apprezzato per la facilità di deploy. A livello enterprise è importante ricordare che, sebbene attraverso un meccanismo che traduce PHP in C chiamato HipHop[21], la rete sociale più grande del mondo[1] ha il proprio codice sorgente in linguaggio PHP.

5.2 Caratteristiche del linguaggio

PHP è un linguaggio di scripting, ovvero un linguaggio di programmazione interpretato. A differenza dei linguaggi di programmazione compilati, che compilano il proprio codice in linguaggio macchina prima dell'esecuzione, il codice PHP viene eseguito per mezzo di un interprete.
In ambito web viene utilizzato attraverso l'uso di un'estensione applicata al web server, per consentire la generazione dinamica di codice HTML. Per Apache, il web server più diffuso, l'estensione è chiamata mod_php.
PHP possiede un gran numero di librerie per eseguire ogni tipo di operazione, dall'elaborazione di immagini alla manipolazione di documenti XML fino alla crittografia.
PHP non si basa su una specifica formale, l'unica documentazione della semantica del linguaggio è data dalla definizione nel codice sorgente dell'implementazione. Tale mancanza rende la modellazione del comportamento del programma in esame complessa per un tool di analisi.
Biggar e Gregg sono gli autori di phc[18], un compilatore alternativo rispetto a Zend che supporta solo codice PHP 4. Durante lo studio della semantica di PHP si sono scontrati con la mancanza di un modello formale per tale linguaggio ed hanno riportato nel proprio studio[26] le caratteristiche che ritenute ambigue e problematiche. Siccome l'analisi della semantica del linguaggio è necessaria anche nei tool di analisi statica, di seguito vengono riportate alcune problematiche da loro evidenziate:

[1] http://www.facebook.com

- *Incongruenze tra PHP 4 e PHP 5*: Le differenze implementative tra PHP 4 e PHP 5 rendono l'analisi statica del codice sorgente problematica. Un caso emblematico riguarda il passaggio di variabili negli argomenti di un metodo che in PHP 4 avveniva di default per valore, in PHP 5 per riferimento. Benchè codice PHP 4 sia ormai datato questa differenza è ancora un vincolo importante per l'esecuzione di un'analisi statica compatibile.

- *php.ini*: Il file di configurazione php.ini influisce sul programma in esame, ad esempio la direttiva *include_path* definisce i files che vengono automaticamente aggiunti alla codebase, mentre *magic_quotes_gpc* cambia il modo secondo il quale le stringhe vengono gestite.

- *Release*: Le nuove release di PHP possono alterare la semantica del linguaggio anche se contengono solo bugfix.

Oltre a queste problematiche la dinamicità del linguaggio comporta la presenza di alcune situazioni difficili da esaminare con la sola analisi statica. Tra queste:

- *Valutazione del codice run-time*: Il costrutto *eval* consente di eseguire come istruzioni le sequenze di caratteri contenute in una stringa come se fossero codice sorgente. Il contenuto di tale stringa però non è conosciuto prima dell'esecuzione del programma, rendendo di fatto impossibile l'analisi di tale contenuto.

- *Inclusione run-time di files esterni*: In PHP è possibile definire l'inclusione di files a seconda dello stato di variabili valutate run-time. Il valore di tali variabili non è definibile tramite analisi statica.

- *Tipizzazione dinamica dei dati*: Il tipo di una variabile non necessita di essere dichiarato e può mutare in modo trasparente a run-time. Questa peculiarità del linguaggio rende complessa la tracciatura dei dati nel flusso di esecuzione poiché possono essere convertiti più volte in modo implicito.

- *Duck typing*: Valori in un oggetto possono essere aggiunti o rimossi in ogni momento dell'esecuzione, facendo si che non risulti possibile predire l'occupazione in memoria dell'oggetto dalla sua dichiarazione.

- *Aliasing*: E' possibile definire alias in grado di referenziare il valore di altre variabili. Tali alias sono mutabili e possono essere creati e distrutti run-time.

- *Variabili di variabili*: Il valore di una stringa può essere usato come indice di un'altra variabile. Ciò è possibile grazie all'uso di una tabella dei simboli al posto di rigide locazioni di memoria.

E' importante poi ricordare come esistano alcune caratteristiche presenti in pressoché tutti i linguaggi che rendono le tecniche di analisi statica molto complesse:

- *Funzioni per la manipolazione di stringhe*: la presenza di una libreria per la manipolazione di stringhe rende complessa l'analisi statica poiché è importante conoscere l'esatto significato di ogni funzione al fine di capire dove e come un valore tainted si può propagare.

- *Espressioni regolari*: la manipolazione di stringhe tramite espressioni regolari complica l'analisi. Teoricamente occorrerebbe comprendere il funzionamento dell'espressione regolare e di conseguenza capire se tale espressione è in grado di agire su valori tainted ed in che modo.

5.3 Modalità di analisi di codice PHP

In letteratura sono presenti diversi approcci all'analisi statica di codice PHP. Come riportato in precedenza, la maggiore differenza riguarda la struttura oggetto in input all'analisi, ovvero quali tipologie di istruzioni vengono processate per ottenere i risultati.
L'input per un tool di analisi statica non è altro che l'output di una delle varie fasi nel processo di interpretazione del codice:

- Codice sorgente
- Lexical analysis
- Syntax analysis
- Bytecode generation

Il codice sorgente non è mai stato utilizzato efficacemente come input nel campo dell'analisi statica. Le regole lessicali che governano il linguaggio rendono estremamente complessa l'interpretazione del codice, che necessita di essere processato per mezzo di espressioni regolari.
Per tale motivo i primi tool che eseguivano analisi statica di codice PHP prendevano in input uno stream di tokens, ovvero una rappresentazione delle istruzioni indipendente dalle regole lessicali del linguaggio. Tale output è frutto della fase di lexical analysis nel processo di interpretazione del codice. Si citano due librerie in grado di convertire da codice sorgente a stream di tokens, *tokenizer*[8], inclusa nella release di PHP e *PHP_TokenStream*[13].

La prima libreria ha un output più semplice, composto da un array di tre elementi: token, numero di riga e corrispondenza in codice PHP, come mostrato nell'esempio sottostante (nel primo listing è riportato il codice sorgente, nel secondo la corrispondente rappresentazione sotto forma di tokens).

```
1  <?php
2  function foo() {
3    if (TRUE) {
4      $foo = TRUE;
5    } else {
6      $foo = FALSE;
7    }
8    return NULL;
9  }
10 ?>
```

```
Array (
  [0] => Array (
    [0] => 368       T_OPEN_TAG
    [1] => <?php\n     text
    [2] => 1         line number
  )
  [1] => Array (
    [0] => 334       T_FUNCTION
    [1] => function
    [2] => 2
  )
  [2] => Array (
    [0] => 371       T_WHITESPACE
    [1] =>
    [2] => 2
  )
  [3] => Array (
    [0] => 307       T_STRING
    [1] => foo
    [2] => 2
  )
  [4] => (
    .
    .
    .
```

La seconda libreria invece fornisce come output una serie di tokens rappresentati come oggetti, riportati nell'esempio sottostante.

```
PHP_Token_Stream Object (
  [flags:SplDoublyLinkedList:private] => 0
  [dllist:SplDoublyLinkedList:private] => Array
  (
    [0] => PHP_Token_OPEN_TAG Object
    (
      [id:protected] => 368
      [text:protected] => <?php\n
      [line:protected] => 1
    )
    .
    .

    [4] => PHP_Token_OPEN_BRACKET Object
    (
      [id:protected] => 501
      [text:protected] => (
      [line:protected] => 2
    )
    .
    .
```

Al termine della conversione, i tool di analisi statica possono così operare su una struttura più semplice da parsare rispetto al puro codice sorgente.

La fase successiva nell'interpretazione di codice PHP consiste nella syntax analysis, ovvero nel parsing per ottenere un abstract syntax tree. Tre librerie sono state individuate per la realizzazione di tale fase: *PHP_Reflection_AST*, *parse_tree*[7] e *PHP-Parser*[9]. Le prime due risultano abbandonate da anni, non supportano quindi i costrutti delle più recenti release di PHP; l'ultima, sebbene rilasciata come alfa, è attualmente l'unica libreria scritta in PHP in grado di generare un abstract syntax tree di codice sorgente basato su PHP 5.3. Esiste poi un ulteriore promettente tool, chiamato *PHP_Depend*[12], che non si occupa di eseguire analisi statica del codice ma effettua analisi delle metriche. Al suo interno contiene una serie di librerie in grado di generare un AST parziale, derivate da *PHP_Reflection_AST*.

L'ultima fase che produce risultati validi per un'analisi statica è la fase di bytecode generation. La libreria che si occupa di produrre tale output è *Bytekit*[1] di Stefan Esser. Oltre a riportare una rappresentazione utente del bytecode generato in fase di compilazione, fornisce anche informazioni sul flusso di istruzioni in esecuzione. L'output di questa fase è però a basso livello e complesso da parsare durante un'analisi statica.

I tools analizzati nel documento corrente utilizzano come input valori provenienti dalle fasi di lexical analysis o syntax analysis, che forniscono una rappresentazione ad alto livello del flusso di istruzioni.

Capitolo 6

Comparazione dei principali tool open source esistenti

> The mantra of any good security engineer is: 'Security is a not a product, but a process.' It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together.
>
> — Bruce Schneier, 'Applied Cryptography' author

Nel seguente capitolo verranno esposte le modalità di funzionamento di alcuni tool che eseguono analisi statica di codice PHP al fine di trovare vulnerabilità. Verranno confrontati gli approcci e verrà valutata la possibile applicabilità durante il normale ciclo di sviluppo software.
Alcuni di questi tool sono effettivamente disponibili, altri sono dei prototipi accademici, altri ancora non supportano le versioni più recenti di PHP. l'obiettivo di tale sezione è quello di illustrare come la ricerca di vulnerabilità tramite analisi statica di codice PHP non presenti soluzioni affermate e si cercherà di spiegarne le motivazioni.

6.1 Pixy

Pixy è un tool di analisi statica scritto in Java per la detection di vulnerabilità in codice PHP 4, rilasciato sotto licenza GPL. Sviluppato da Jovanovic[32] come lavoro accademico, inizialmente consentiva la ricerca di sole vulnerabilità di tipo XSS ma successivamente è stato esteso alla ricerca di altre vulnerabilità di tipo taint-style, come SQL injections e command injections.
Pixy è flow sensitive, ovvero prende in considerazione l'ordine delle istruzioni del programma e la scansione che effettua è di tipo globale (interprocedural analysis), ovvero non valuta una funzione singolarmente ma tiene in considerazione il contesto in cui essa viene eseguita. Inoltre Pixy è dotato di un meccanismo di analisi degli alias che consente di ridurre i numerosi falsi positivi che tale caratteristica del linguaggio, quando utilizzata,

potrebbe generare.
Pixy è disponibile anche in una versione web-based a funzionalità limitate sul sito ufficiale[14].

La scansione di Pixy si basa sulla definizione di tre parametri:

- *Gli entry points del programma*: GET, POST e COOKIE.
- *Le funzioni di sanitizzazione*: htmlentities, htmlspecialchars ed opportuno type casting.
- *Sensitive sinks*: funzioni che ritornano output al browser, come print, echo, printf.

Queste definizioni sono stabilite tramite un file di configurazione in formato testuale.
L'obiettivo della scansione consiste nel determinare in quali circostanze è possibile che un dato tainted possa raggiungere un sensitive sink senza essere sanitizzato in modo opportuno. La tecnica per determinare ciò è quella della data-flow analysis, ovvero si computano in ogni punto del programma i possibili contenuti delle variabili per tracciare quali possono essere tainted. La data-flow analysis opera sul control flow graph del programma, quindi è necessario che prima dell'esecuzione venga costruito un albero (nello specifico un parse-tree) del file PHP in input. Tale operazione viene svolta con Jflex[5], un lexical analyzer scritto in java e Java Parser Cup[3], un generatore di parser in java.
Prima di eseguire la taint analysis sul control flow graph risultante vengono eseguite diverse operazioni, tra cui l'*alias analysis*, che fornisce informazioni sugli alias, e la *literal analysis*, che valuta le condizioni di branch ed esclude i percorsi che non possono essere eseguiti a run-time. Entrambe queste operazioni servono ad aumentare la precisione del tool, sebbene ne aumentino la complessità.
Pixy ignora le sanitizzazioni custom, generalmente effettuate con espressioni regolari, per due motivi: prima di tutto non è in grado di valutarle, cosa che è possibile solo con un'analisi dinamica, inoltre si ritiene a priori che effettuare sanitizzazioni mediante l'uso di espressioni regolari sia una pessima decisione implementativa a causa della facilità di errore.

La grossa problematica che affligge Pixy è data dal fatto che è stato abbandonato nel 2007, in questo momento non è in grado di parsare codice scritto secondo canoni moderni. Le ingenti modifiche apportate al linguaggio dalla release di PHP 5 hanno reso Pixy inadeguato per gli attuali progetti scritti in PHP.
La mancanza di una solida community ha fatto si che nessuna terza parte abbia forkato il progetto per proseguirne lo sviluppo, rendendolo ormai troppo datato.

Secondo i risultati dei test eseguiti dall'autore, Pixy ha una percentuale di falsi positivi che si aggira intorno al 50%, soglia considerata dall'autore stesso accettabile per un tool di analisi statica su un linguaggio fortemente dinamico come PHP. Secondo le analisi eseguite da de Poel[29] nel 2010, durante la scansione del codice sorgente di alcuni progetti open source viene dimostrato che Pixy non è in grado di funzionare con codebase recenti; la scansione ha infatti riportato risultati per solo cinque su sedici dei software analizzati e, neppure in questi casi, i risultati sono stati attendibili.

6.2 Saner

Saner è un prototipo, basato su Pixy, che combina le peculiarità di un'analisi statica con quelle di un'analisi dinamica. Sviluppato da Balzarotti et.al.[24] dimostra come sia possibile combinare i due approcci di analisi per migliorare i risultati ottenuti.
Saner esegue, in ordine, prima un'analisi statica sul sorgente, dopodiché procede con l'analisi dinamica di un data flow particolare. La parte di analisi statica viene eseguita da Pixy, che si occupa di determinare in che modo l'applicazione processa l'input, individuando eventuali sanitizzazioni mancanti o incomplete. Successivamente interviene una fase di analisi dinamica che si occupa di ricostruire il codice responsabile per la sanitizzazione dell'input. Una volta ricostruito, Saner processa tale codice con input malevoli per individuare eventuali problemi nelle sanitizzazioni.
Facendo affidamento sull'analisi dinamica, Saner è in grado di valutare eventuali sanitizzazioni custom al posto di reputarle direttamente come inaffidabili, cosa che il solo Pixy era costretto a fare.
Nell'esempio seguente viene evidenziato il valore del contributo dell'analisi dinamica nella scelta della giusta decisione da intraprendere. Si suppone di dover analizzare il seguente codice:

```
1 $input = $_GET['x'];
2
3 $standard = htmlentities($input);
4 $standard = 'Hello' . $standard;
5 echo $standard;
6
7 $custom = str_replace('<', '', $input);
8 $custom = 'Hello' . $custom;
9 echo $custom;
```

Come Pixy, la maggior parte dei tool di analisi statica avrebbero marchiato la prima parte del codice riportato come safe a causa della presenza della funzione *htmlentities*, che solitamente fa parte delle funzioni di sanitizzazione. Nel secondo caso però è presente una sanitizzazione custom costruita con la funzione *str_replace*. Siccome *str_replace* non fa parte delle funzioni di sanitizzazione, il codice viene marchiato come unsafe ed un warning viene riportato all'utente. Saner consente di eseguire tale sanitizzazione in modo dinamico sfruttando input appositamente malevoli al fine di verificare l'efficacia della sanitizzazione stessa, riducendo il numero di warning.
In un certo senso, Saner automatizza le operazioni che uno sviluppatore dovrebbe fare per verificare se un warning riportato da un tool di analisi statica è effettivamente una problematica di sicurezza.

I risultati riportati dall'autore dimostrano come la fase di analisi dinamica aumenti in modo elevato l'affidabilità del tool. Nell'analisi di cinque applicativi open-source di media complessità, Pixy segnala l'incapacità di determinare il valore della procedura di sanitizzazione per ben 66 sinks. Con l'ausilio dell'analisi dinamica è stato possibile determinare

l'effettiva presenza di 14 vulnerabilità, mentre per i restanti 52 sinks l'analisi non è stata in grado di determinare dei valori di input che bypassassero la sanitizzazione.

6.3 RIPS

RIPS[28] è un tool per l'analisi statica di codice PHP rivolto alla ricerca di vulnerabilità sviluppato da Johannes Dahse. E' l'unico tool open source scritto in PHP considerato usabile in sviluppo in data attuale.
Il tool è in grado di individuare vulnerabilità di tipo XSS, SQL Injection, file disclosure, code evaluation, remote command execution e busines logic flaw attraverso dei rule-sets definiti nei propri file di configurazione.
RIPS lavora in due fasi, costruzione del modello e analisi. Nella fase di costruzione del modello si possono individuare le operazioni di analisi semantica e lessicale, parsing e control flow analysis. Nella fase di analisi si individuano le operazione di taint analysis e local and global analysis.

La fase di analisi lessicale e semantica è caratterizzata dall'uso della libreria built-in *tokenizer* per trasformare i costrutti del linguaggio in uno stream di tokens, attraverso l'uso della funzione *token_get_all()* che trasforma ogni istruzione del linguaggio in un array costituito dal token, dal numero di riga dell'istruzione e dal codice originale. Successivamente vengono rimossi i tokens ritenuti inutili per l'analisi, come le spaziature ed il codice HTML, e trasformati alcuni tokens in equivalenti strutture più semplici da parsare (ad esempio vengono convertiti i costrutti di branches realizzati mediante la forma compatta in classici costrutti if-else).
La fase successiva, quella di parsing, avviene attraverso l'analisi dello stream di tokens, una sola volta per ogni file al fine di garantire migliori performance. In questa fase viene creata per ogni file una lista delle dipendenze e per alcuni costrutti vengono eseguite operazioni particolari. In caso di *T_INCLUDE* ad esempio lo stream di tokens del file da includere viene aggiunto allo stream corrente, in caso di *T_VARIABLE* viene controllato lo scope della variabile, ovvero se è una variabile globale o locale, e di conseguenza viene aggiunta ad una lista.
La fase di control flow analysis avviene in modo estremamente diverso da Pixy: se nel tool di Jovanovic si faceva riferimento al control flow graph, in RIPS la generazione dello stesso è stata evitata per problemi di performance. RIPS esegue la control flow analysis sfruttando la presenza di alcuni tokens come le parentesi graffe, *T_EXIT* e *T_THROW* che danno indicazioni sulle uscite da flussi di istruzioni.

La fase di analisi si basa sulle direttive definite attraverso array PHP nei file di configurazione. Tali direttive indicano gli input, i sinks, le funzioni di sanitizzazione con i rispettivi parametri da tracciare ed i token da ignorare.
RIPS cerca chiamate a funzioni, controlla se la funzione trovata è nella lista dei sinks ed in caso affermativo valuta i parametri di tale funzione. Una volta individuati i parametri rilevanti controlla nelle liste precedentemente create di variabili globali e locali se tali possono essere o meno tainted. In caso affermativo viene riportata la potenziale

vulnerabilità.
RIPS è in grado di determinare se una funzione definita dall'utente è un sink. Nel caso in cui ciò accada infatti l'algoritmo può risalire ai parametri di input e quindi comprendere se il sink è determinato da uno di essi. In caso affermativo oltre alla funzione stessa anche le chiamate vengono trattate come un sink.

RIPS possiede una interfaccia web configurabile che consente di lanciare la scansione e che riporta i risultati. Tale interfaccia consente di definire per quale tipologia di vulnerabilità effettuare la scansione, il livello di verbosity e se comprendere le sottodirectory o meno. Il report dei risultati è dettagliato, vengono mostrate le linee di codice sorgente che riportano problematiche comprensive di motivazione, è possibile visualizzare un grafico delle chiamate che devono essere effettuate per raggiungere un certo sink e viene utilizzata la syntax highlighting per mostrare le linee di codice coinvolte.

Nei test effettuati alla Ruhr-University Bochum su una piattaforma ad uso interno, RIPS è stato in grado di esaminare 16870 linee di codice in 84 files, per un totale di 181 funzioni in 2.3 secondi. Nei risultati sono stati individuati diversi falsi positivi, dovuti all'uso di espressioni regolari per la sanitizzazione, metodologia non contemplata nell'analisi effettuata da RIPS, ed all'uso di fwrite per scrivere su file di testo. A seconda del livello di verbosity utilizzato varia il numero di falsi positivi riscontrati, poichè vengono incluse nei risultati delle vulnerabilità sospette ma non verificate.

RIPS è un tool che ben si adatta all'analisi statica per la ricerca di vulnerabilità. E' veloce, è configurabile con diversi livelli di verbosità, possiede una semplice e comprensibile interfaccia per il report dei risultati e produce un basso numero di falsi positivi.
Sebbene possieda le classiche limitazioni dei tool di analisi statica, come l'incapacità di valutare stringhe generate a run-time nel caso di inclusioni dinamiche di files o nel caso di chiamate dinamiche a funzioni che ne riducono l'accuratezza, l'estrema velocità rende adatto il suo utilizzo all'interno nel ciclo di sviluppo software. RIPS è un tool in sviluppo ed attualmente non supporta completamente la semantica di PHP, ad esempio alcuni costrutti particolari come variabili di variabili o l'aliasing. Inoltre non supporta i costrutti tipici di PHP 5, ma supporta solo in modo basilare la dichiarazione di classi, oggetti e metodi.

L'approccio proposto da un tool come RIPS è estremamente differente da quello proposto da Pixy. RIPS è stato implementato con l'obiettivo di essere veloce e di riportare in modo chiaro i risultati, spiegando come una vulnerabilità funziona al posto di semplicemente specificare la linea in cui essa si trova, Pixy invece possiede un meccanismo di scansione più accurato ed ha come obiettivo il riuscire a riportare il minor numero di falsi positivi in assoluto.

Capitolo 7

Vulture

Vulture[16] è un prototipo di tool per analisi statica volta alla ricerca di vulnerabilità di applicazioni web, progettato in collaborazione con EURECOM durante lo svolgimento di questo lavoro di tesi.
Nato inizialmente con l'obiettivo di produrre un sistema in grado di rilevare vulnerabilità di tipo Http Parameter Pollution server side, è stato generalizzato al fine di costruire una piattaforma estensibile, alla quale lo sviluppatore può aggiungere i propri insiemi di regole.
Inizialmente si è pensato di lavorare all'estensione di un tool esistente, in modo da aggiungere solo la parte di codice riguardante Http Parameter Pollution. Purtroppo però i candidati, analizzati in precedenza in questo documento, presentavano problematiche di progettazione rilevanti, che non potevano essere risolte facilmente ma avrebbero comportato la riscrittura di grosse porzioni del tool stesso. Le caratteristiche individuate come fondamentali erano le seguenti:

- Supporto a PHP 5
- Possibilità di modifica della codebase
- Estensibilità
- Velocità di scansione
- Basso numero di falsi positivi
- Supporto
- Scritto in PHP

La possibilità di modifica della codebase implicava che il tool fosse open source, gli unici due tools opensource rilevanti sono Pixy e RIPS. Pixy non supporta però PHP 5, non è supportato dall'autore o da una community, non presenta una rilevante velocità di scansione ma è in grado di riportare un basso numero di falsi positivi grazie al supporto agli alias. RIPS invece non supporta gli alias, caratteristica non molto utilizzata in PHP, e non fornisce come Pixy pieno supporto a classi, metodi ed oggetti, introdotti da PHP 5. Nonostante ciò è veloce e supportato dall'autore, caratteristiche fondamentali per la

scelta di un tool sul quale lavorare.
Purtroppo però la codebase di RIPS si è rivelata decisamente problematica da estendere e modificare, il codice sorgente in formato procedurale con funzioni da migliaia di righe poco commentate hanno reso di fatto più semplice la riscrittura del tool rispetto al refactoring del codice esistente. Oltretutto RIPS possiede un approccio all'analisi basato sulla funzione tokenizer di PHP, ritenuta ad un livello troppo basso, ovvero troppo vicino al codice sorgente per l'analisi statica.

La soluzione ideale è stata individuata in un tool in grado di generare e parsare l'abstract syntax tree del codice sorgente, ottenendo una rappresentazione ad alto livello sulla quale effettuare la scansione. Il tool deve essere veloce, estensibile affinché sia possibile per uno sviluppatore aggiungere delle regole personalizzate per la ricerca di vulnerabilità. Quest'ultima caratteristica porta ad una considerazione: il tool verrà usato per trovare vulnerabilità nel codice sorgente scritto in PHP, quindi da sviluppatori che normalmente lavorano con tale linguaggio. Se si vuole che il tool venga poi esteso dalla community, difficilmente si potrà attenere un riscontro se lo scrive in un linguaggio diverso da PHP, proprio perché le competenze di chi lo userà sono focalizzate su quel linguaggio.

Vulture è un tool scritto in PHP dotato di un'interfaccia web. Non necessita di database e l'unico requisito per utilizzarlo è un web server. Il tool è basato su Symfony 2[15], un framework per la realizzazione di applicazioni web scritto in PHP da Fabien Potencier. Tale scelta è stata effettuata poiché tale framework fornisce le più comuni caratteristiche di un'applicazione web, come il meccanismo di routing delle pagine e la separazione tra logica e presentazione.
L'utilizzo di Symfony 2 non comporta un overhead per la visualizzazione delle pagine ed evita che si debba reinventare la ruota per problematiche già affrontate e di scarso interesse ai fini dello sviluppo del tool di analisi statica.

Vulture si presenta con una schermata7.1 nella quale viene richiesto di inserire il percorso alla directory contenente il codice sorgente da analizzare.

Ogni file viene esaminato in tale directory, sottodirectory comprese.
Successivamente il tool genera un AST del codice sorgente fornito in input. La generazione dell'AST avviene per mezzo di un progetto open source sperimentale chiamato PHP-Parser[9]. Sebbene in fase alfa, è attualmente l'unico strumento scritto in PHP in grado di generare un AST di codice sorgente che supporta tutte le caratteristiche dell'attuale release di PHP.
Esistono soluzioni alternative abbandonate o incomplete, come php-sat[10], fermo dal 2006, o PHP_Depend[12], che comprende un parser attualmente incompleto.

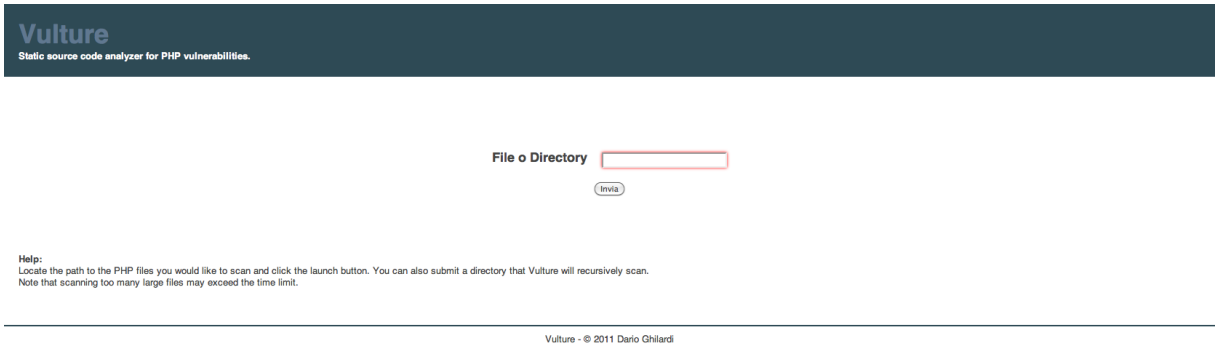

Figura 7.1: Vulture - Schermata iniziale

L'output di PHP-Parser è una rappresentazione ad alto livello del codice sorgente, sotto forma di una combinazione di array ed oggetti.
Sono stati effettuati alcuni test per determinare l'efficacia di PHP-Parser: la scansione completa della codebase di Symfony2 funziona correttamente e richiede circa 35 secondi, un ottimo risultato viste le dimensioni della codebase in esame.

```
array(
    0: Stmt_Echo(
        exprs: array(
            0: Scalar_String(
                value: This is
                a php code example.<br />
                isBinary: false
                type: 1
            )
        )
    )
    1: Stmt_Echo(
        exprs: array(
            0: Expr_ArrayDimFetch(
                var: Variable(
                    name: _SERVER
                )
                dim: Scalar_String(
                    value: PHP_SELF
                    isBinary: false
                    type: 0
                )
            )
```

```
24          )
25      )
26      2: Stmt_Echo(
27          exprs: array(
28              0: Expr_Concat(
29                  left: Scalar_String(
30                      value: And this is a GET variable
31                      :
32                      isBinary: false
33                      type: 1
34                  )
35                  right: Expr_ArrayDimFetch(
36                      var: Variable(
37                          name: _GET
38                      )
39                      dim: Scalar_String(
40                          value: p
41                          isBinary: false
42                          type: 0
43                      )
44                  )
45              )
46          )
47      )
```

Dopo la generazione dell'AST, si utilizza una metodologia di traversing dello stesso, identificando gli elementi utili a seconda della vulnerabilità che si sta cercando. Per determinare quali essi siano si usano i bundle di Symfony: la definizione di un nuovo bundle con una determinata struttura di files viene riconosciuta automaticamente come la definizione di un set di regole per la ricerca di una nuova vulnerabilità.
Ogni bundle costruito per Vulture consente la definizione di un set di input, un set di sinks ed un set di funzioni di sanitizzazione. E' su questi tre parametri che Vulture basa la sua scansione, in modo similare a quanto eseguito da RIPS.

Durante la scansione Vulture salva in memoria le variabili che vengono a contatto con valori tainted, ovvero alle quali vengono assegnati valori provenienti dall'input. Tali variabili vengono custodite in un array, che viene aggiornato ad ogni istruzione processata con le eventuali nuove variabili che divengono tainted a causa di assegnamenti. Se si incontra una funzione di sanitizzazione applicata su una variabile di tipo tainted, Vulture rimuove la variabile dall'array. Se invece si incontra un sink con una variabile presente nell'array tainted Vulture segnala una nuova vulnerabilità. Al fine di poter determinare se una funzione di sanitizzazione è efficace non basta che si identifichi la variabile come parametro della stessa, occorre conoscere anche in quale parametro viene posizionata: le funzioni in PHP sono state definite senza una cognizione, capita che il parametro sul quale viene applicata la funzione sia in prima posizione in alcuni casi, in seconda in altri. Per tale motivo è necessario definire un vocabolario di funzioni con le rispettive posizioni per i parametri in input, al fine di poter determinare se il parametro è posizionato o meno nel punto corretto per la sanitizzazione. Questo caos è una delle caratteristiche che hanno contribuito a ritenere PHP un linguaggio poco strutturato e l'hanno portato ad essere

valutato negativamente dalle community di sviluppatori di altri linguaggi[11].
Quando vengono incontrate delle inclusioni, Vulture apre il file collegato e lo include nel flusso corrente costruendo un flusso unico continuo sul quale eseguire la scansione.

Attualmente il tool non è usabile e completo. Costruire un tool di analisi statica da zero è un problema complesso che non poteva essere risolto unicamente nel solo corso di questa tesi. Tool come RIPS e Pixy hanno richiesto anni di sviluppo (per la precisione RIPS è in sviluppo da 3 anni, Pixy è stato in sviluppo per altrettanti), e sono comunque loro stessi incompleti per quanto riguarda il supporto a classi, oggetti ed ai restanti costrutti di PHP 5. Nonostante ciò sono state progettate le fasi successive per lo sviluppo di Vulture:

- *Loop e branches*: Vulture non supporterà loop e branches al fine di rendere la scansione più veloce possibile.
- *Sanitizzazioni custom*: Vulture non supporterà sanitizzazioni custom, frutto ad esempio di espressioni regolari. Sarà però flessibile, in quanto consentirà la definizione manuale di funzioni di sanitizzazione, utile nel caso in cui si utilizzino dei framework che forniscono funzioni aggiuntive oltre a quelle del linguaggio. E' ovviamente un arma a doppio taglio, definire una funzione errata come sanitizzazione custom può risultare disastroso quindi occorre la massima cautela.
- *Inclusioni e chiamate a funzioni dinamiche*: Vulture, come tutti i tools di analisi statica, non è in grado di valutare inclusioni o chiamate a funzioni dinamiche. In questo caso verrà riportata una segnalazione.
- *Aliasing delle variabili*: L'aliasing delle variabili non è una caratteristica molto utilizzata in PHP. Realizzare un sistema in grado di valutare l'applicazione di tale funzionalità nell'analisi statica è molto complesso, ed il valore di ritorno sarebbe comunque basso, valido solo per le applicazioni che sfruttano tale caratteristica. Per tali ragioni lo sviluppo di tale funzionalità è stato posticipato.
- *Definizioni di classi e metodi e relative chiamate*: Il supporto a PHP 5 non è stato attualmente realizzato da alcun tool. Vulture si pone questo obiettivo per avere il pieno supporto alle applicazioni PHP moderne. L'idea alla base della realizzazione di tale funzionalità consiste nel trattare in modo particolare le classi con tutta una serie di accorgimenti, senza snaturare il funzionamento del tool. Sarà prevista una libreria di supporto che si occuperà di riconoscere i costrutti tipici delle classi e dei relativi oggetti e metodi.

Oltre alle caratteristiche analizzate fin'ora, PHP 5.3, un'ulteriore evoluzione del linguaggio, ha introdotto alcune novità come i namespace, le closures ed il late static binding. Tali strutture non sono attualmente previste nell'implementazione di Vulture ma sono senza dubbio fondamentali affinchè diventi possibile effettuare la scansione di moderne applicazioni web. Per tale motivo questo è una delle problematiche su cui sarà importante concentrarsi in futuro.

Capitolo 8

Discussione

In questo lavoro di tesi sono state analizzate dal punto di vista implementativo le principali soluzioni open source per l'esecuzione di analisi statica rivolta alla ricerca di vulnerabilità in applicazioni web. Esistono ovviamente anche soluzioni commerciali, tra le quali HP Fortify Static Code Analyzer[4] ed Armorize CodeSecure[2]. Questi due prodotti non sono rivolti esclusivamente all'analisi di codice sorgente scritto in PHP ma ognuno di essi supporta molteplici linguaggi. Entrambi i tool sono stati confrontati nell'eccellente lavoro di analisi dei risultati svolto da Nico de Poel[29].
E' da notare come i tool open source non abbiano catturato un'audience rilevante. Sebbene il lavoro per la realizzazione sia tutt'altro che indifferente, i programmatori PHP hanno dimostrato scarso interesse per questo genere di tool, perlomeno non sufficiente per la nascita di una community di supporto. Ciò è senza dubbio la concausa dell'abbandono di tool promettenti come Pixy, insieme al mancato supporto per PHP 5.
Generalmente questa è la situazione che si verifica con tool che nascono in ambito accademico, i quali inizialmente raccolgono interesse, ma poi vengono abbandonati dai loro autori quando sono costretti a spostarsi verso altri topic di ricerca. Il risultato porta ad avere molti approcci diversi e molti prototipi, ma alla fine nessun prodotto sufficientemente valido per essere usato in produzione.
Per quanto riguarda i tool commerciali, non è possibile determinare la validità dell'approccio utilizzato poiché sono a sorgente chiuso. De Poel ha eseguito un'ottima comparazione basandosi sui risultati ottenuti dai tool open source e da quelli commerciali derivandone che i prodotti commerciali sono gli unici maturi sul mercato per questa tipologia di analisi.
E' importante però sottolineare come le carenze in questo ambito non siano da imputare solamente ai tool presenti sul mercato. La problematica maggiore è certamente la scarsa educazione degli sviluppatori alle problematiche di sicurezza. Se un numero maggiore di sviluppatori usasse questa tipologia di tool nel loro ciclo di sviluppo software, probabilmente tool come Pixy non sarebbero attualmente abbandonati, ma avrebbero raggiunto un grado di sviluppo eccellente per il problema che ambiscono a risolvere.
La questione si può riconoscere in una mancanza di tipo formativo, non tutti gli sviluppatori possiedono conoscenze sufficienti a sviluppare software sicuro; allo stesso modo non sono solitamente richieste certificazioni che determinino la conoscenza o meno delle problematiche di sicurezza nel software per uno sviluppatore. Si potrebbe formare

maggiormente il personale, con evidenti costi da affrontare e responsabilità aggiuntive da stabilire, oppure assegnare a terzi il lavoro di code review. Quest'ultima soluzione è la più applicata, ma porta ad un ciclo di sviluppo software sbilanciato, in cui una vulnerabilità viene segnalata in un momento tardivo dello stesso sviluppo, aumentandone i costi di correzione.
Se le esigenze economiche sono da sempre la priorità, appare incomprensibile la mancanza di tool specifici che approccino questa problematica durante lo sviluppo software. E' per questo possibile ritenere che ci sia un enorme fetta di mercato scoperta per un possibile tool di analisi statica con i predetti scopi, perché è un'esigenza che le software house possiedono, ma di cui effettivamente sembrano non accorgersi.

Capitolo 9

Conclusioni

L'analisi statica finalizzata alla ricerca di vulnerabilità è una tecnologia promettente per ridurre le problematiche di sicurezza nelle applicazioni web moderne. La velocità di esecuzione consente l'integrazione di tale tecnica all'interno del ciclo di sviluppo software, facilitando il riconoscimento di vulnerabilità ed annullando la finestra di esposizione in quanto si va a sostituire il meccanismo di riconoscimento a posteriori con un sistema di integrazione continua. Ciò contribuisce anche ad abbassare i costi di gestione grazie all'immediata risposta fornita dal tool. Per tali motivi l'analisi statica, sebbene imprecisa, può aggiungersi alla review del codice e fornire un grosso contributo nello sviluppo di applicazioni web sicure.
La creazione di un tool di analisi statica non è però un compito semplice, specialmente per linguaggi estremamente dinamici come PHP. L'impossibilità di valutare i tipi di dato, le inclusioni dinamiche e le chiamate a funzioni dinamiche sono solo alcune delle problematiche che non è possibile risolvere con la sola analisi statica e che influiscono pesantemente sui risultati delle scansioni.
Nonostante queste evidenti limitazioni, il settore dell'analisi statica è un ramo quasi inesplorato ed estremamente promettente, che potrebbe fare la fortuna di chi si dedicherà alla produzione del primo tool in grado di supportare i costrutti di PHP 5 e potrebbe essere un enorme passo avanti in una community di sviluppatori da sempre poco attenti alle problematiche di security.

Bibliografia

[1] Bytekit. https://github.com/Tyrael/bytekit.

[2] Codesecure. http://www.armorize.com/codesecure/.

[3] CUP: LALR Parser generator in Java. http://www2.cs.tum.edu/projects/cup/.

[4] Fortify Source Code Analyzer. https://www.fortify.com/products/hpfssc/source-code-analyzer.html.

[5] Jflex: The fast scanner generator for Java. http://jflex.de.

[6] Martin Fowler: Continuous integration. http://martinfowler.com/articles/continuousIntegration.html.

[7] Parse_Tree. http://pecl.php.net/package/parse_tree.

[8] PHP built-in tokenizer library. http://php.net/manual/en/book.tokenizer.php.

[9] PHP-Parser. https://github.com/nikic/PHP-Parser.

[10] php-sat: Static Analysis for PHP. http://www.program-transformation.org/PHP/PhpSat.

[11] PHP sucks but it doesn't matter. http://www.codinghorror.com/blog/2008/05/php-sucks-but-it-doesnt-matter.html.

[12] PHP_Depend. http://http://pdepend.org.

[13] PHP_tokenstream. https://github.com/sebastianbergmann/php-token-stream.

[14] Pixy official website. http://pixybox.seclab.tuwien.ac.at/pixy/webinterface.php.

[15] Symfony Framework. http://symfony.com/.

[16] Vulture: Github repository. https://github.com/ingo86/vulture.

[17] Flawfinder. http://www.dwheeler.com/flawfinder/, 2010.

[18] phc: Documentazione. http://www.phpcompiler.org/doc/latest/grammar.html, 2010.

[19] PQL: A language for expressing patterns of events on objects. It provides a frontend to static and dynamic program analyses to go find those sequences on the program as it runs. `http://pql.sourceforge.net/`, 2010.

[20] Splint. `http://www.splint.org/`, 2010.

[21] HipHop for PHP: transforms PHP source code into highly optimized C++. `https://github.com/facebook/hiphop-php`, 2011.

[22] PHP: hypertext Preprocessor. `http://www.php.net`, 2011.

[23] Marco Balduzzi. HTTP Parameter Pollution Vulnerabilities in Web Applications. In *BlackHat Europe 2011*, 2011.

[24] D. Balzarotti, M. Cova, V. V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *In Proceedings of the 2008 IEEE Symposium on Security*, 2008.

[25] Tim Berners Lee, Roy Fielding, and Larry Masinter. RFC 3986, Uniform Resource Identifier (uri): Generic Syntax. `http://datatracker.ietf.org/doc/rfc3986/`, 2005.

[26] Paul Biggar and David Gregg. Static analysis of dynamic scripting languages, 2009. Draft.

[27] Cigital. ITS4: A Static Vulnerability Scanner for C and C++ Code. `http://www.cigital.com/its4/`, 2010.

[28] Johannes Dahse. RIPS - A static source code analyser for vulnerabilities in PHP scripts. `http://php-security.org/downloads/rips.pdf`. Ruhr-University Bochum.

[29] Nico L. de Poel. Automated Security Review of PHP Web Applications with Static Code Analysis. Master's thesis, May 2010.

[30] Stefano di Paola and Luca Carrettoni. HTTP Parameter Pollution. `https://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf`, 2009.

[31] Fortify. RATS: Rough Auditing Tool for Security. `http://www.fortify.com/security-resources/rats.jsp`, 2010.

[32] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *2006 IEEE Symposium on Security and Privacy*, 2006.

[33] OWASP. Top Ten Project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2010.

[34] Donn B. Parker. *Fighting Computer Crime*. John Wiley & Sons, Inc., 1998.

[35] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. In *Transactions of the American Mathematical Society*, pages 358–366, 1953.

[36] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.

Elenco delle figure

Elenco delle tabelle