
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

# Prefazione

Testo della prefazione.

# Indice

# Capitolo 1

# Introduzione

Testo dell'introduzione.

# Capitolo 2

# Motivazione

Testo della motivation.

# Capitolo 3

# 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 malintenzionati, per tale motivo è nata l'esigenza di meccanismi che potessero garantire la sicurezza dei dati.

Un'applicazione web consiste in un software, posizionato su un server, al quale si accede tramite un'interfaccia web. Tipicamente 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 serie di richieste 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 costruisce di conseguenza delle risposte coerenti con le richieste. Questa operazione viene effettuata dalle istruzioni definite nell'applicazione, create tramite il linguaggio server-side utilizzato. Tale linguaggio 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 però 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 degli attacchi alla sicurezza di un'applicazione sono rivolti all'errata implementazione del software oppure a vulnerabilità create dalla 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.

## 3.1 Fondamenti di sicurezza

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

- Confidentiality: indica la misura che vieta la 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 è stata messa in discussione la definizione di sicurezza attraverso questi tre termini, con la proposta di termini aggiuntivi. Ad esempio nel 2002 Donn Parker[**?**] propose un'alternativa composta da sei termini, aggiungendo ai tre classici principi le nozioni di possession, authenticity e utility. 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.

## 3.2 Vulnerabilità nelle applicazioni web

Le vulnerabilità sono presenti nel software per vari motivi: per errate decisioni architetturali ed implementative, per mancata conoscenza da parte dello sviluppatore delle problematiche di security e per 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 a chiunque 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 determinate certificazioni per essere praticato.
Un'altra motivazione che esclude la sicurezza dal processo di sviluppo software è costituita dalle condizioni economiche, le quali possono incidere sulle tempistiche e quindi restringere 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 altamente focalizzato sull'adesione ai requisiti imposti in fase di progettazione, molto meno sulle implicazioni che può avere una errata implementazione delle specifiche. Solitamente anche in presenza di vulnerabilità non si violano le 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. E' proprio in questo codice che statisticamente si concentrano le vulnerabilità, quelle in cui la sicurezza non è un requisito.
Un esempio di tale situazione è 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 ed è passato ad una porzione del programma vulnerabile (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 ed accetta solo input da provenienti da sinks che vengono validati attraverso espressioni regolari. L'interprete quindi considera sicuro solo un parametro che viene sanitizzato in questo modo.
Questa tecnica ha un 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. [**?**]). 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), 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, un progetto rilasciato con cadenza triennale che raccoglie 10 vulnerabilità tipiche di applicazioni web. Gli obiettivi di OWASP sono:

- 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-practice e tool.
- 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 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 classica vulnerabilità negli applicativi professionali).

### 3.2.1 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 SQL injection, LDAP injection e OS injection.
E' molto comune trovare questa tipologia di vulnerabilità in codice legacy, ovvero non più supportato dal produttore, 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.
Un esempio di tale vulnerabilità può essere il seguente:

```
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 preso 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.

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 quella di effettuare l'escape dei caratteri speciali usando specifiche sintassi per ogni interprete.

### 3.2.2 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 validazione o escaping.
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 tipologie di cross site scripting:

- Stored: Il codice viene iniettato nel server in modo permanente, in un database, in un forum, in un commento, ecc. La vittima ottiene lo script malevolo ad ogni visita della pagina.
- 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 viene convinto a cliccare su tale link il codice viene eseguito.
- DOM based: Un payload malevolo viene eseguito come risultato della modifica del DOM[1] 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 effettuate al DOM.
  E' diverso da stored e reflected XSS poichè in questo caso il payload melevolo non è nella pagina di risposta del server ad una richiesta.

[1] Document Object Model

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 attivo del browser. E' quindi necessario effettuare l'escape dei contenuti in input basati su codice HTML, a seconda del contesto in cui tali dati verranno poi utilizzati.

### 3.2.3 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 il management delle sessioni e dell'autenticazione utente.

### 3.2.4 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, ad 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.

### 3.2.5 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 img apposito e fare in modo che la vittima lo visualizzi da loggata.
L'esempio sottostante mostra un tag img 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.

### 3.2.6 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, devono essere inoltre 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 accedute 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 l'implementazione delle 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 o siti contenenti malware.

## 3.3 Security development life cycle

Tutte le realtà che si occupano di sviluppo software implementano un modello che definisce le fasi entro cui lo sviluppo prende parte. Tali fasi sono definite nel software development life cycle (SDLC).
Esistono diversi modelli che implementano il 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 i test). Per tali motivi è opportuno che l'analisi di sicurezza venga implementata nello sviluppo, introducendo un diverso software development life cycle.

INSERIRE QUI IMMAGINE SDLC

L'immagine **??** 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 la finestra di esposizione, presente 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 una rapidità di scansione e feedbacks immediati. Per questo compito l'analisi statica è altamente efficiente.

# Capitolo 4

# 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. Tale 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.

## 4.1 Storia

Il primo approccio all'analisi statica è storicamente da attribuire al tool di Unix *grep* (General Regular Expression Print), che ricerca in uno o più file di testo le linee che corrispondono ad uno o più modelli specificati con espressioni regolari o stringhe letterali e produce un elenco di linee in cui è stata trovata una corrispondenza. 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 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 costo ha la sua correzione.

## 4.2 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[1] 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.

[1] Integrated Development Environment

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[2], 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 real-time alla lentezza di un'esecuzione alla ricerca di problematiche di sicurezza. 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. Tuttavia il principale scopo dell'analisi statica è quello di aiutare lo sviluppatore nel comprendere il codice e nel trovare e risolvere problemi, siano essi di sicurezza o meno.

## 4.3 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

[2]http://www.doxygen.org

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 possono individuare 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[3] 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 di codice. L'approccio definito dall'analisi dinamica è particolarmente adatto per il testing ed il debugging.

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. **??**, 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.

[3]Source Code Management System

# Capitolo 5

# Applicazione dell'analisi statica alla sicurezza di applicazioni web

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, ITS4 e Flawfinder, 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 (abstract syntax tree) 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. Al fine 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, aumentando il costo per la risoluzione di eventuali problematiche.

## 5.1 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

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 preprocessa il codice per ottenere una rappresentazione ad un livello più basso, sfruttando le normali operazioni eseguite dal compilatore:

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[1].
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.

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

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 grammaticale 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 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 raggiungere una rappresentazione da analizzare. In base al dato che si analizza esistono innumerevoli approcci alla fase di analisi, ognuno con le proprie caratteristiche e di conseguenza con risultati differenti.

—RIVEDERE—

Brian Chess at.al.[**?**] hanno proposto un approccio composto da due fasi, *intraprocedural analysis* o *local analysis* ed *interprocedural analysis* o *global analysis*. La prima si occupa di analizzare una funzione individualmente, la seconda di analizzare le interazioni tra le funzioni.
Analizzare una funzione individualmente significa tenere traccia delle proprietà dei dati nella funzione e delle condizioni per le quali una funzione può essere chiamata in modo sicuro. Tracciare tali proprietà diventa però problematico in caso di loop e branches, la quantità di dati diventa ingente per cui l'approccio risulta difficilmente 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 poichè 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. Quando viene eseguita l'analisi si tengono in considerazione solo tali elementi per determinare gli effetti che hanno sull'intero programma.

—RIVEDERE—

# Capitolo 6

# Analisi statica di codice PHP

# Capitolo 7

# Comparazione dei principali tool esistenti

## 7.1 Pixy

## 7.2 Saner

## 7.3 RIPS

# Capitolo 8

# Vulture

## 8.1 Problematiche

## 8.2 Sviluppi futuri

# Capitolo 9

# Discussione

# Capitolo 10

# Conclusioni

# Bibliografia


[1] Marco Balduzzi, Christian Platzer, Thorsten Holz, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. Abusing Social Networks for Automated User Profiling. In *Preceeding of the 13th international conference on Recent advances in intrusion detection*, pages 422–441, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All Your Contacts Are Belong to Us : Automated Identity Theft Attacks on Social Networks. In *Preceedings of the 19th international conference on World Wide Web (WWW)*, 2009.

[3] B. Bowen, S. Hershkop, A. Keromytis, and S. Stolfo. Baiting inside attackers using decoy documents. *Security and Privacy in Communication Networks*, pages 51–70, 2009.

[4] Monica Chew and Ben Laurie. ( Under ) mining Privacy in Social Networks. In *Preecedings of Web 2.0 Security and Privacy Workshop (W2SP)*, 2008.

[5] Catherine Dwyer and Starr Roxanne Hiltz. Trust and privacy concern within social networking sites : A comparison of Facebook and MySpace. In *Preceedings of the Thirteenth Americas Conference on Information Systems (AMCIS)*, 2007.

[6] Brad Fitzpatrick and David Recordon. Thoughts on the Social Graph. `http://bradfitz.com/social-graph-problem`, 2007.

[7] Ralph Gross, Alessandro Acquisti, and H John Heinz Iii. Information Revelation and Privacy in Online Social Networks ( The Facebook case ). *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2005.

[8] Miguel Helft. Google thinks it knows your friends. `http://bits.blogs.nytimes.com/2007/12/26/google-thinks-it-knows-your-friends/`.

[9] Facebook Inc. Facebook privacy guidelines. `http://www.facebook.com/policy.php`.

[10] Jesper M. Johansson and Roger Grimes. Security-through-obscurity. `http://technet.microsoft.com/en-us/magazine/2008.06.obscurity.aspx`.

[11] Aleksandra Korolova. Privacy Violations Using Microtargeted Ads : A Case Study. 2010.

[12] W. Mackay. Triggers and barriers to customizing software. In *Preceedings of CHI'91*, pages 153–160. ACM Press, 1991.

[13] Kevin D. Mitnick. *The art of deception.* John Wiley & Sons, Inc., 2002.

[14] Nick Nikiforakis, Marco Balduzzi, Steven Van Acker, Wouter Joosen, Davide Balzarotti, and Sophia Antipolis. Exposing the Lack of Privacy in File Hosting Services. *Sophia*, 2010.

[15] J. Yuill, M. Zappe, D. Denning, and F. Feer. Honey- files: deceptive files for intrusion detection. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop*, pages 116–122, 2004.

[16] Mark Zuckerberg. Making control simple. `http://blog.facebook.com/blog.php?post=391922327130`.