POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

SCRIPTSHARK: ANALISI SEMI-STATICA DEL CODICE JAVASCRIPT PER LA PROTEZIONE DA ATTACCHI INFORMATICI

Relatore: Prof. Stefano ZANERO

Tesi di Laurea di:

Riccardo ZUCCHINALI

Matricola n. 754854

Anno Accademico 2010–2011

Ringraziamenti

Desidero ringraziare profondamente la mia famiglia, Dimitri, Simona e Anna, per avermi sostenuto pazientemente negli anni mentre raggiungevo questo traguardo. Ringrazio anche tutti i parenti e gli amici di famiglia che, a vario titolo, mi hanno sempre spronato a raggiungere risultati sempre migliori.

Un ricordo speciale a chi non c'è più e che sarebbe sicuramente orgoglioso di me.

Un sentito ringraziamento a Federico e Stefano, per avermi assistito, con notevole pazienza, durante lo sviluppo della tesi.

Ringrazio inoltre tutti i miei amici che hanno sopportato negli anni la mia scarsa presenza sociale e che d'ora in poi non si sentiranno più rispondere "non posso, ho un'esame". Un ringraziamento particolare a Luca, collega di corso sopravvissuto negli anni alle mie tristi battute e co-autore di inenarrabili ludici passatempi universitari.

Infine, una menzione speciale all'insostituibile sostegno di qualsiasi programmatore o sistemista informatico: il caffè.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco delle abbreviazioni

DBMS Database Management System

HTML HyperText Markup Language

FN False Negative

FP False Positive

TP True Positive

OO Object Oriented

JIT Just-in-Time

DOM Document Object Model

CSS Cascading Style Sheet

HTTP HyperText Transport Protocol

AJAX Asyncronous JavaScript And XML

AST Abstract Syntax Tree

XSS Cross-Site Scripting

CMS Content Management System

Introduzione

Il linguaggio JavaScript riveste un ruolo da protagonista nel Web moderno. La potenza e la versatilità che lo caratterizzano sono fondamentali nelle pagine HTML per realizzare efficaci animazioni grafiche e nelle piu' complesse applicazioni Web per realizzare interfacce utente complesse e reattive attraverso sistemi dinamici di caricamento asincrono dei dati (come, ad esempio, AJAX).

Purtroppo la flessibilità stessa del linguaggio e la particolarità del contesto di riferimento, quale è il browser Web, rendono JavaScript pericoloso, in quanto potenziale veicolo per la realizzazione di attacchi informatici. Drive-by-download, ovvero l'attacco mirato alla compromissione del client mediante il download e l'esecuzione di un file malevolo all'insaputa dell'utente sfruttandone i privilegi, clickjacking, ovvero la sovrapposizione grafica di elementi della pagina, volti al furto di informazioni, frame-busting, ovvero la replica di risorse valide al fine di ingannare l'utente, sono i principali attacchi realizzabili sfruttando JavaScript e la loro pericolosità è amplificata dall'ampia superficie d'attacco disponibile (ovvero chiunque si connetta al Web attraverso un browser in grado di interpretare JavaScript, ovvero *tutti* gli utenti del Web).

La ricerca accademica, unitamente ad alcune aziende impegnate sul fronte della sicurezza informatica, hanno realizzato nel tempo sistemi capaci di analizzare intere pagine al fine di identificare potenziali minacce, integrando poi queste caratteristiche in sistemi di pubblico dominio al fine

di impedire all'utente finale di raggiungere tali risorse corrotte, raggiungendo lo scopo finale di proteggere la navigazione (come, ad esempio, il sistema Google SafeBrowsing).

I sistemi esistenti coprono quasi completamente le opzioni possibili per l'analisi, con tecniche sia statiche che dinamiche di analisi del codice, in alcuni casi ibride, e tutti, in generale, fermano il loro livello d'analisi al singolo URL. Questo non inficia assolutamente l'efficacia di questi sistemi, ma rende difficoltosa una analisi specificatamente basata sul singolo frammento di codice.

L'obiettivo di questa tesi consiste nel realizzare uno strumento di protezione completo che sfrutti una metodologia di rilevazione delle minacce ancora poco esplorata, realizzando e valutando l'efficacia di un sistema d'analisi principalmente statico per il codice JavaScript basato sul singolo frammento di codice, volto anche alla costruzione di una knowledge base di codice JavaScript malevolo.

Abbiamo innanzitutto sviluppato un componente plug-in per il browser Mozilla Firefox, in grado di intercettare e bloccare il codice JavaScript prima della sua effettiva esecuzione, al quale abbiamo affiancato un server centrale in grado di raccogliere, analizzare ed identificare il codice JavaScript. Con la stessa estensione browser abbiamo inoltre costruito un database di codice JavaScript grezzo, utile per addestrare e valutare l'efficacia del sistema di identificazione. Per concludere, abbiamo valutato le performance complessive del sistema di protezione, valutandone l'impatto nell'utilizzo reale.

La presentazione del lavoro svolto e dei risultati conseguiti è strutturata come segue:

- Capitolo 1: il linguaggio JavaScript, breve storia, peculiarità, utilizzo tipico e le potenziali minacce note.

- Capitolo 2: stato dell'arte dei sistemi d'analisi per il codice JavaScript e dei sistemi di protezione esistenti per la navigazione Web.
- Capitolo 3: il sistema Scriptshark, architettura, componenti fondamentali.
- Capitolo 4: valutazione efficacia del sistema di identificazione minacce, costruzione del dataset, addestramento classificatore.
- Capitolo 5: valutazione performance del sistema completo di protezione, valutazione impatto sull'esperienza di navigazione.

Capitolo 1

JavaScript

Le origini di JavaScript risalgono all'anno 1996 circa, quando nei laboratori Netscape Communications inizia la progettazione di un linguaggio di programmazione multipiattaforma, nome in codice "Mocha", da poter distribuire congiuntamente alle pagine composte con l'HyperText Markup Language (HTML) e che fosse possibile eseguire direttamente dal browser Web, per spostare parte del carico di elaborazione della nascente Internet dai server ai client. A quell'epoca infatti, l'elaborazione dei contenuti delle pagine era completo appannaggio dei server Web e la vertiginosa espansione della rete e del bacino d'utenza stava generando un innalzamento di carico non sopportabile nel lungo periodo.

JavaScript raggiunge lo stadio di maturità intorno al 1997, quando acquisisce definitivamente il nome che conosciamo tutt'ora e viene parallelamente reso standard nelle specifiche ECMAScript; il suo sviluppo è proseguito costantemente nel tempo fino ai giorni nostri, raggiungendo la versione 5.1 (corrente).

1.1 Caratteristiche del linguaggio

Il linguaggio deriva sintatticamente dal C, con il quale presenta infatti alcune similitudini.

Contrariamente a quanto istintivamente si può essere portati a pensare, JavaScript non presenta legami con il linguaggio Java, pur nascendo all'incirca nello stesso periodo e risultando sintatticamente simile (questo perchè anche Java deriva sintatticamente dal linguaggio C).

JavaScript è, per definizione, un linguaggio interpretato, debolmente tipizzato e debolmente orientato agli oggetti. Nella pratica risulta essere un tipico linguaggio di *scripting*, nel quale le variabili non hanno tipo, se non una distinzione grezza tra numerico, stringa e oggetto, e nel quale è possibile progettare con il paradigma Object Oriented (OO), pur non avendo supporto completo all'ereditarietà ed al concetto di interfacce.

Il linguaggio nasce infatti non per essere un vero e proprio linguaggio completo, ma per poter operare in modo semplice e diretto con il Document Object Model (DOM), il modello run-time a oggetti del documento HTML renderizzato nel browser, e per essere eseguito e gestito direttamente dal browser stesso, prescindendo quindi dalla piattaforma del client. Queste caratteristiche rappresentano tutt'oggi i cardini fondamentali dell'utilizzo di JavaScript: infatti, sebbene lo standard HTML permetta l'utilizzo di vari linguaggi di scripting all'interno delle pagine, JavaScript rappresenta ad oggi lo *standard de facto* per questo particolare ambito.

Nell'ultimo periodo, il crescente ricorso all'utilizzo di JavaScript per costruire complesse applicazioni Web, ha portato alla luce il suo punto debole principale, ovvero il fatto di essere un linguaggio interpretato con scarse performance globali. L'evoluzione tecnologica è riuscita però a superare il problema, realizzando i compilatori Just-in-Time (JIT), capaci di tradurre in tempo reale il codice JavaScript direttamente in linguaggio macchina, eseguibile dal sistema in modo efficiente e, soprattutto, altamente

performante.

1.2 Utilizzo tipico

Nato per consentire l'esecuzione di brevi procedure direttamente nel browser del client, inizialmente JavaScript ha avuto fortuna come motore per la realizzazione di semplici animazioni ed effetti grafici interattivi nelle pagine HTML, all'epoca statiche e rigide nella loro struttura. Per la prima volta si potevano infatti realizzare pagine capaci di sostituire intere parti di contenuto senza effettuare una nuova richiesta completa al server, a tutto vantaggio dell'efficienza e dell'esperienza utente.

Successivamente, l'introduzione di nuove tecnologie per la definizione grafica della presentazione HTML, quali i Cascading Style Sheet (CSS) e la nuova filosofia di definizione delle pagine attraverso il markup semantico, hanno permesso a JavaScript di essere il regista indiscusso di animazioni e interfacce grafiche reattive in-browser sempre piu' complesse e accattivanti.

Negli ultimi tempi inoltre, JavaScript è sfruttato in modo importante anche per il caricamento selettivo ed incrementale di contenuti all'interno delle pagine stesse, realizzando il cosiddetto paradigma Asyncronous JavaScript And XML (AJAX). Con questo approccio è possibile, pilotando tutte le operazioni attraverso JavaScript stesso, caricare dinamicamente in modo asincrono dati all'interno della pagina HTML visualizzata dal browser, sfruttando il protocollo HTTP per la comunicazione con il sistema server ed utilizzando invece il formato XML per la formattazione dei dati in transito. Il paradigma AJAX rappresenta una delle rivoluzioni concettuali e tecnologiche degli ultimi anni ed ha permesso lo sviluppo di vere e proprie applicazioni Web che propongono un sistema di interfacce grafiche e una reattività globale così simili alle classiche applicazioni standalone

classiche tali da soppiantarle in taluni casi in modo completo, aprendo la strada all'utilizzo del browser non solo come strumento per la navigazione del Web ma come vera e propria interfaccia di accesso ad applicazioni complesse, come ad esempio la suite di applicazioni per l'ufficio di Google Documents, che offre, interamente attraverso il browser, una piattaforma per la creazione e gestione di documenti funzionalmente pari ai classici pacchetti applicativi standalone come Apache OpenOffice e similari.

1.3 Javascript come veicolo di minacce

JavaScript è un linguaggio potente e molto versatile al tempo stesso, interpretato ed eseguito interamente all'interno del processo del browser. Il linguaggio infatti, come già accennato, è in grado di operare in modo nativo sul DOM, il modello della pagina visualizzata nel browser. Se nell'accezione originale, questa capacità dava la possibilità allo sviluppatore di rendere interattiva la pagina Web, nel proseguo degli anni sono emersi invece gli aspetti negativi e dannosi di questa potenzialità.

JavaScript viene utilizzato attivamente per realizzare moltissime tipologie di attacchi Web, tra i quali troviamo:

- Cross-Site Scripting (XSS): questi attacchi mirano all'iniezione di codice JavaScript malevolo all'interno di risorse valide attraverso l'utilizzo di sistemi di interazione con l'utente, come bacheche per commenti e fanno parte della macro-categoria di attacchi informatici legati alla mancanta o errata validazione dei dati di input.
  Se il sistema di Content Management System (CMS) che governa il sito Web in esame non è sufficientemente sofisticato nel trattare i dati ricevuti dagli utenti, un malintenzionato può infatti iniettare del codice JavaScript malevolo, formattandolo nel classico formato HTML, all'interno di un contenuto della pagina, ad esempio tramite un com-

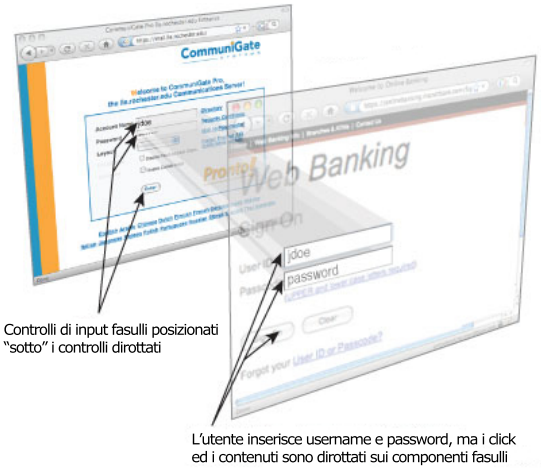

Figura 1.1: Esempio di un attacco clickjacking

mento, facendo in modo che tutti i visitatori successivi di quella risorsa siano esposti all'attacco.

- Abuso di funzionalità: in questa macro-categoria rientrano le tipologie di attacco che sfruttano tecniche e funzionalità perfettamente lecite per realizzare comportamenti malevoli.
  In questa categoria rientrano gli attacchi di tipo clickjacking [1], letteralmente "dirottamento dei clic", del quale è possibile vedere uno schema elementare in Figura 1.1, volti a costruire nella pagina dei componenti reattivi, quali pulsanti, link, campi di testo, graficamente occultati in sottofondo ad altri componenti regolari: con questo stratagemma il malintenzionato può recuperare informazioni riservate o invocare procedure senza che l'utente se ne accorga "dirottando" per l'appunto il contenuto o l'azione compiuta sul componente regolare della pagina verso la sua controparte nascosta.
  Un altro esempi di attacco simile è il frame-busting, ovvero l'abuso del componente `iframe` nei documenti HTML: attraverso questo componente infatti il malintenzionato può incorporare attivamente

una risorsa all'interno di un'altro documento, che verrà poi manipolato dal malintenzionato per reagire in modo diverso all'interazione con l'utente. Così facendo l'attaccante può costruire pagine che *appaiono* graficamente perfette alle loro controparti regolari, pur non essendolo, ingannando l'utente e riuscendo tipicamente in un furto di informazioni riservate.
In base all'obiettivo dell'attacco, si possono quindi avere varie sottocategorie specifiche:

- Furto d'informazioni: attacchi mirati alla sottrazione di informazioni riservate dell'utente, come potrebbero esserlo le credenziali di accesso ad una sistema di home-banking o di un social network.
- Contraffazione di contenuti: attacchi mirati alla replica di risorse lecite all'interno di altre pagine non riferite alla stessa organizzazione al fine di ingannare l'utente.
- Redirezione involontaria: attacchi di questo tipo mirano a generare delle redirezioni involontarie del browser direttamente o indirettamente, ad esempio attraverso catene di redirezioni consecutive, verso un'altra risorsa malevola. Nella maggior parte dei casi le redirezioni sono attivate in modo autonomo, senza un'intervento esplicito dell'utente, benchè i browser integrino al loro interno specifiche policy per ridurre fortemente il fenomeno delle azioni JavaScript non esplicitamente invocate dall'utente.

- Sfruttamento vulnerabilità: l'attacco mira allo sfruttamento di particolari vulnerabilità software insite nel browser stesso o in uno dei suoi componenti esterni per eseguire un pacchetto malevolo con gli stessi privilegi operativi del browser o, peggio, più elevati. L'attacco

è quindi volto alla compromissione del client utente e l'esempio più diffuso appartenente a questa categoria è il drive-by-download.

La nostra attenzione si concentra in particolare su questi ultimi, ovvero sul codice JavaScript che realizza lo sfruttamento delle vulnerabilità del browser per prendere il controllo del client attraverso l'iniezione di codice malevolo.

1.3.1 Attacchi mirati alla compromissione del client

In questa tipologia di attacchi, JavaScript viene utilizzato solamente come strumento per realizzare l'attacco. Rispetto ad alle altre tipologie di attacco, gli attacchi mirati al controllo del client sono tecnologicamente più complessi, in quanto sfruttano specificatamente particolari vulnerabilità software, insite nel browser o in un suo plug-in, per operare l'attacco stesso.

Questi rende il codice stesso dell'attacco più corposo: lo sfruttamento delle vulnerabilità, infatti, richiede tipicamente un elaborato sistema di analisi dell'ambiente di esecuzione, volto ad identificare la presenza degli specifici componenti che verranno compromessi durante l'attacco, e una altrettanto elaborata fase di realizzazione del contesto necessario ad esporre la vulnerabilità al fine sfruttarla attivamente.

La maggior parte di questi attacchi però viene rilasciata in formato offuscato: con questa tecnica, il codice malevolo che realizza realmente l'attacco viene nascosto, con metodi di cifratura e compressione, in una stringa di caratteri priva di significato. La riconversione del codice offuscato in codice eseguibile avviene solamente durante l'esecuzione, attraverso una specifica procedura che è integrata nella fase di analisi dell'ambiente: solo se l'ambiente è effettivamente attaccabile, la procedura eseguira il deoffuscamento mettendo in luce il codice d'attacco. In questo modo l'attaccante evita l'esposizione inutile del proprio codice critico in sistemi per i quali può

intuire a priori il fallimento, rendendo difficoltosa l'analisi della specifica vulnerabilità sfruttata e la sua relativa correzione.

```
1  ...
2  function BAlrZJkW ( pvOWGrVU , Hhvo4b_X ) {
3          while ( pvOWGrVU . length *2< Hhvo4b_X ) pvOWGrVU += pvOWGrVU ;
4          pvOWGrVU = pvOWGrVU . substring (0, Hhvo4b_X /2);
5          return pvOWGrVU ;
6  }
7
8  function Exhne69P () {
9          if (! z5AsUJQZ ) {
10                 var MSnMnmRB = 0 x0c0c0c0c ;
11                 var YuL42y0W =
12                 unescape ("\% u9090 \% u9090 ...\% u3030 \% u3030 \%
                       u3030 \% u3030 \% u3038 \% u0000 ");
13                 ...
14                 var pvOWGrVU = unescape ("\% u0c0c \% u0c0c ");
15                 pvOWGrVU = BAlrZJkW ( pvOWGrVU , Hhvo4b_X );
16                 for (var cYQZIEiP =0; cYQZIEiP < cFyP_X9B ; cYQZIEiP ++) {
17                         RBGvC9bA [ cYQZIEiP ]= pvOWGrVU + YuL42y0W ;
18                 }
19         ...
20         }
21 }
22
23 function a9_bwCED () {
24         try {
25                 var OBGUiGAa = new ActiveXObject ('Sb. SuperBuddy ');
26                 if ( OBGUiGAa ) {
27                         Exhne69P ();
28                         dU578_go (9);
29                         OBGUiGAa . LinkSBIcons (0 x0c0c0c0c );
30                 }
31         } catch (e) { }
32         return 0;
33 }
34 ...
```

Listato 1.1: Esempio di attacco JavaScript de-offuscato

Un esempio di attacco JavaScript deoffuscato è visibile nel Listato 1.1:

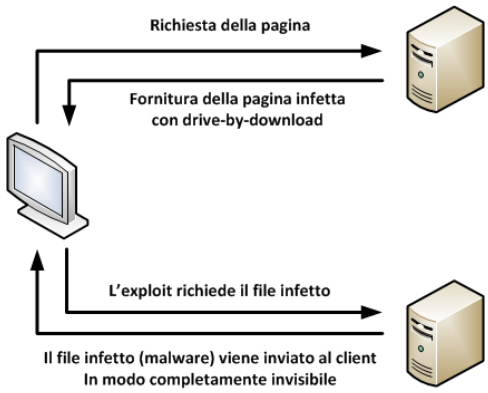

Figura 1.2: Schema di un attacco drive-by-download

si può notare, nell'ultima funzione, l'istanziazione di un componente ActiveX e l'utilizzo di un suo metodo, chiamato con un argomento opportunamente valorizzato, al fine di sfruttarne la vulnerabilità e consentire quindi l'esecuzione del codice malevolo precedentemente caricato in memoria. Le prime due funzioni sono invece dedite alla costruzione, e relativa valorizzazione, in memoria della struttura contenente il codice binario da iniettare, proveniente da una stringa codificata in Unicode.

L'attacco di questo tipo più diffuso è il drive-by download: il codice iniettato nel client risulta essere una procedura di download e successiva messa in esecuzione di un file, in modo completamente silenzioso e senza che l'utente possa effettivamente accorgersi dell'azione in corso. A quel punto la macchina è ormai compromessa, in quanto viene mandato in esecuzione un eseguibile di dubbia provenienza e con scopi tutt'altro che leciti. Con questa tipologia di attacco si verifica la distribuzione di malware o, in generale, la diffusione di infezioni informatiche volte alla costruzione di botnet, il tutto sfruttando un vasto bacino d'utenza.

Capitolo 2

Stato dell'arte

La protezione nella navigazione Web è uno dei principali problemi moderni sui quali sia l'attività di ricerca accademica che quella delle aziende impegnate nel settore pongono la loro attenzione. Nel nostro caso specifico, ci occuperemo in particolar modo degli attacchi di tipo drive-by download, ovvero quegli attacchi mirati alla compromissione del client attraverso l'iniezione di codice malevolo.

Il problema della protezione di un sistema informatico genera storicamente due problemi distinti ma fortemente correlati:

- Identificazione delle minacce
- Protezione dalle minacce

L'identificazione è il primo passo fondamentale per la costruzione di un sistema di protezione: senza la capacità di poter discriminare tra ciò che rappresenta un contenuto lecito e ciò che rappresenta invece una minaccia, nessun sistema di protezione può operare correttamente. L'identificazione deve necessariamente sfruttare un'analisi condotta sul codice, che, di qualunque tipo esso sia, è sempre possibile su due fronti:

- Fronte statico: viene analizzato il codice nella sua forma grezza, pura. Si può procedere analizzando testualmente il codice oppure usan-

do altri programmi capaci di simulare completamente l'esecuzione in modo simbolico (esecutori simbolici).

- Fronte dinamico: il codice viene mandato in esecuzione realmente, avendo l'accortezza di isolare l'ambiente di esecuzione in modo da impedire la fuoriuscita dell'attacco oltre l'ambiente di test stesso (sandboxing) oppure attraverso l'uso di sistemi capaci di gestire passo-passo l'esecuzione in profondità, impedendo gli attacchi mentre si stanno per consumare.

Il problema principale dell'analisi dinamica è l'eccessivo spreco di risorse: oltre all'onere computazionale legato alla necessità di eseguire realmente il codice per valutarne gli effetti in una macchina appositamente configurata e monitorata, al fine di isolarla ed impedire che eventuali minacce si propaghino da essa, non bisogna dimenticare l'assoluta necessità di ripristinare, dopo ogni analisi, l'immagine iniziale di tale macchina, in modo da minimizzare la sovrapposizione di attacchi ed effetti della precedente esecuzione. Questo porta il tempo reale d'analisi intorno alle decine di minuti, un onere complessivamente accettabile solo da organizzazioni medio-grandi, che dispongono di una vasta potenza computazionale (come ad esempio Google con il suo servizio SafeBrowsing).

L'analisi statica risulta quindi globalmente più veloce e meno impegnativa in termini di risorse, ma perde sull'effettiva efficacia dell'analisi: nel caso specifico degli attacchi JavaScript, la maggior parte del codice malevolo risulta infatti offuscato, compresso e cifrato, in un modo tale da non essere immediatamente leggibile nel codice statico. Questo complica il processo d'analisi, perché lo limita e impone l'utilizzo di strumenti d'analisi più complessi, come ad esempio gli esecutori simbolici e gli analizzatori sintattici, che risultano comunque essere non completamente esaustivi riguardo al problema, in quanto comunque impossibilitati ad analizzare l'eventuale codice dinamico generato dal codice stesso.

Identificare le minacce rappresenta quindi, di per sé, un problema importante, a volte risolvibile solo parzialmente. Spesso infatti non è possibile discriminare fortemente tra codice lecito e codice malevolo, in virtù del fatto che molto raramente la minaccia si compone di elementi specificatamente riconoscibili come tali ed è l'insieme di codice,contesto e sistema a determinare la pericolosità reale di uno specifico frammento di codice. Il processo di identificazione poggia quindi, nella maggior parte dei casi, su sistemi di tipo euristico o statistico per la classificazione degli elementi, che utilizzano i risultati dell'analisi come riferimento per riconoscere eventuali minacce, con gli ovvi problemi legati ai tassi di falsi negati e falsi positivi raggiunti dal sistema complessivo.

Il sistema di identificazione singolarmente non è però un sistema fine a sé stesso, non è infatti in grado di offrire un servizio diretto all'utenza finale: è necessario un impianto tecnologico capace di identificare ed intercettare le minacce nei contesti reali e agire di conseguenza, al fine di prevenire o bloccare eventuali attività illecite. Non sempre questo risulta però possibile. Ad esempio, se il sistema da proteggere opera in un contesto di reattività elevata, il processo di protezione potrebbe rallentare l'attività iniziale fino a renderla inutilizzabile; similmente la protezione potrebbe essere tecnologicamente così invasiva nei confronti dell'attività da proteggere da renderla non fruibile.

2.1 Principali sistemi d'analisi esistenti

Il mondo accademico ha sempre avuto particolare attenzione riguardo i problemi di sicurezza legati all'utilizzo improprio del codice JavaScript ed al Web in generale e si è impegnata a fondo per sviluppare sistemi in grado di identificare le minacce legate all'esecuzione del codice all'interno delle pagine HTML.

Il lavoro ha consentito di riconoscere come gli attacchi vengono effettuati e quali componenti deboli vengono maggiormente colpiti dalle pratiche malevole. Si è riusciti così a scoprire che gli attacchi JavaScript sono tipicamente attacchi offuscati, nei quali il codice d'attacco non è esposto direttamente nello script infetto bensì incorporato in un pacchetto dati adeguatamente offuscato, con tecniche di cifratura e codifica, decifrabile solo dallo script iniziale in ben determinate condizioni d'esecuzione e la sua messa in esecuzione successiva si basa fondamentalmente sul sistema di esecuzione dinamica di JavaScript.

```
1 ...
2 var _0xc90d=["\x68\x72\x65\x66","\x6C\x6F\x63\x61\x74\x69\x6F\x6E",
3         "\x68\x74\x74\x70\x3A\x2F\x2F\x77\x77\x77\x2E\x70\x6F\x6C
4         \x69\x6D\x69\x2E\x69\x74"];
5 function redirector(_0x6eb2x2){document[_0xc90d[1]][_0xc90d[0]]
6         =_0x6eb2x2;} ;redirector(_0xc90d[2]);
7 ...
```

Listato 2.1: Esempio di codice JavaScript offuscato

Una caratteristica accomuna tutti i sistemi di analisi ed identificazione delle minacce JavaScript esistenti ad ora: il livello di granularità dell'analisi medesima.
Tutti i sistemi presentati di seguito sono basati sull'analisi del contenuto completo relativo a singoli URL e la loro capacità di discriminare tra risorse malevoli e non rimane a quel livello: nessun sistema ad ora è in grado di analizzare il singolo frammento di codice JavaScript determinandone il grado di pericolosità in modo puntuale.

2.1.1 WEPAWET

M.Cova et al. in [2] propongono WEPAWET, un sistema nato con l'obiettivo di identificare principalmente gli attacchi di tipo drive-by-downloads

ed in seguito utilizzato anche per l'identificazione generica di codice JavaScript malevolo.

E' un sistema dinamico, ovvero l'analisi viene effettuata eseguendo realmente il codice da analizzare in un ambiente protetto e gestito (chiamato JSAND), al fine di tracciare le operazioni critiche ed identificare pattern di potenziali attacchi noti, come l'anomala presenza di un elevato numero di redirect e di specifici componenti ActiveX istanziati, o la costruzione di stringhe estremamente lunghe o dal contenuto sospetto, che potrebbe rappresentare dello shellcode.

Il sistema è, al momento, attivo e di utilizzo pubblico, erogato come servizio Web, accessibile all'url `http://wepawet.iseclab.org/`.

2.1.2 NOZZLE

Benjamin Livshits et al. hanno sviluppato NOZZLE [3], un sistema specifico per l'identificazione di attacchi heap-spraying all'interno del browser, ovvero attacchi che mirano ad utilizzare lo spazio heap come area per iniettare il codice malevolo da eseguire. In JavaScript il processo è tipicamente realizzato concatenando ripetutamente una stringa con sè stessa fino al limite massimo consentito dall'interprete, ottenendo una grande area di memoria compatta allocata interamente nello heap.

Il sistema di analisi opera dinamicamente come processo separato dal browser, cercando attivamente all'interno della memoria allocata da JavaScript e dal browser stesso eventuali tracce di istruzioni x86. Nel caso vengano effettivamente trovate delle sequenze valide, il sistema recupera l'interno contenuto del codice riconosciuto e procede alla verifica di pericolosità, attraverso l'analisi dei blocchi NOP presenti.

Il sistema infatti non analizza nel dettaglio tutte le istruzioni, ma si concentra solo sulle istruzioni che non interferiscono con l'esecuzione e lo stato della macchina: in ogni shellcode infatti è sempre presente una zona usata

come destinazione del salto d'istruzione necessario all'esecuzione del codice malevolo. Tali zone non possono essere offuscate e, pertanto, il sistema le può identificare facilmente, a differenza invece del codice d'attacco reale che può essere offuscato o realizzato in modo tale da non suscitare sospetti sul reale compito dello stesso.

2.1.3 ZOZZLE

Benjamin Zorn e i suoi colleghi presso il Microsoft Reaserch Lab, propongono ZOZZLE [4], che risulta essere uno dei pochi sistemi che consente l'analisi principalmente statica del codice, al fine di identificare il codice malevolo.

ZOZZLE realizza un completo sistema di identificazione e protezione dal codice malevolo, mediante l'analisi statica del codice e l'utilizzo di un classificatore per identificare la presenza di pattern ricorrenti riferibili ad attacchi noti.

Il sistema di analisi non è completamente statico, ma è, riutilizzando la specifica dicitura usata dagli autori stessi per descriverlo, *principalmente statico*. Il sistema, pur essendo di base statico, quindi rivolto all'analisi del codice nella sua forma originale e non attraverso l'utilizzo di sistemi di esecuzione dinamica in sand-box e similari, è però in grado di analizzare anche il codice che JavaScript può *dinamicamente* generare durante l'esecuzione. In questo modo può superare il problema legato all'impossibilità di analizzare il vero codice d'attacco nel caso di minacce offuscate.

L'analisi si estende quindi oltre il solo codice statico, tracciando anche eventuale codice generato dinamicamente, sfruttando un analizzatore d'albero sintattico (AST). Il processo d'analisi è volto all'estrazione di un insieme di caratteristiche, quali pattern ricorrenti di attacchi noti come l'utilizzo di specifici componenti ActiveX da parte di JavaScript, piuttosto che cicli e strutture di controllo volte alla generazione in memoria di vaste aree

potenzialmente destinate al contenimento di shellcode, che verranno poi utilizzate dal classificatore di tipo Bayesiano, per determinare se l'oggetto d'analisi è, oppure no, una risorsa maligna.

2.1.4 ROZZLE

Ancora Benjamin Zorn ed il suo gruppo di ricerca dei Microsoft Research Lab propongono ROZZLE [5], un sistema dinamico per l'identificazione di codice malevolo generico.

Il sistema si basa sull'osservazione che, se una pagina contiene del codice lecito, eventuali variazioni del contesto di esecuzione (ad esempio la versione del browser o la presenza, o meno, di un determinato plug-in installato) produrranno costantemente i medesimi risultati. Una pagina contenente invece del codice malevolo potrebbe reagire al cambiamento di contesto, modificando il risultato finale della propria esecuzione, ad esempio non proseguendo nell'attacco se l'ambiente non è idoneo, oppure causando eccezioni in esecuzione.

Il sistema opera quindi eseguendo più volte il contenuto di una determinata pagina, variando il contesto stesso d'esecuzione e segnalando come potenziali malevoli i contenuti che reagiscono in modo anomalo (ad esempio, abortendo l'esecuzione di propria iniziativa o in seguito ad un errore grave) al cambiamento di contesto.

2.2 Principali sistemi di protezione esistenti

Di pari importanza dei sistemi d'analisi ed identificazione delle minacce risultano essere le tecnologie ed i sistemi creati al fine di proteggere l'utenza finale dalle minacce annidate nelle pagine Web.

Benchè gli attacchi di tipo drive-by-download siano a tutti gli effetti attacchi mirati alla compromissione del client utente mediante l'esecuzio-

ne di eseguibili malevoli, i classici sistemi commericiali di antivirus non si sono rivelati robusti rispetto a questa particolare minaccia. La sofisticazione sempre più importante degli attacchi, in particolare al riconoscimento del contesto del client, per verificare la presenza o meno di determinati software in esecuzione, unita a sistemi di protezione direttamente integrati nel browser, come ad esempio le tecniche di sand-boxing per i processi di rendering delle pagine, hanno superato ben presto le capacità dei sistemi antivirus generali, spingendo invece la ricerca verso sistemi di protezione più specifici.

2.2.1 Google SafeBrowsing

Un sistema più recente e globalmente molto diffuso è quello di Google SafeBrowsing, che impedisce attivamente all'utente di arrivare materialmente sulle risorse che sono note essere malevole.

Per implementare questa tipologia di sistema è però necessario che il browser sia modificato in modo tale da consultare, prima di collegarsi effettivamente al server remoto che rappresenta il fornitore di contenuti per la risorsa in esame, una blacklist, al cui interno sono memorizzati gli indirizzi delle risorse che sono state segnalate come malevoli: in base alla risposta ottenuta, ovvero in base alla presenza o meno in tale lista della risorsa esaminata, il browser proseguirà nel recupero dei contenuti oppure bloccherà l'accesso, avvisando l'utente della minaccia e dei potenziali pericoli annessi.

Il sistema è efficace e molto diffuso: tutti i maggiori browser attualmente in circolazione adottano questo sistema di protezione od uno che eroga un servizio equivalente.

Lo svantaggio principale di questo sistema ed altri similari è principalmente quello del mantenimento operativo della blacklist. In particolar modo risulta fondamentale il continuo processo di ricerca ed aggiornamento

necessario al sistema di identificazione delle minacce per risultare sempre efficace, oltre alla pura necessità di indicizzare periodicamente le risorse disponibili sul web per aggiornare attivamente la blacklist e per rimuovere da essa eventuali risorse che non presentano più pericolosità. In particolare Google utilizza un sistema proprietario di identificazione delle minacce, basato sull'analisi dinamica del comportamento di una macchina virtuale in seguito alla visualizzazione della risorsa da esaminare.

2.2.2 ICESHIELD

ICESHIELD [6] rappresenta un approccio innovativo al problema della protezione dal codice JavaScript.

Esso sfrutta lo stesso contesto JavaScript di esecuzione e le proprietà specifiche del linguaggio per inserire un controllore di alto livello all'interno dell'esecutore stesso del codice. Il sistema risulta essere in grado di identificare anomalie durante le chiamate ai metodi o l'esecuzione di particolari routine e neutralizzarne gli effetti, il tutto senza introdurre overhead esterni ottenendo quindi performance assolute molto elevate. Inoltre il sistema è a tutti gli effetti implementato in JavaScript stesso, e può essere "iniettato" nelle pagine da proteggere in modo trasparente ed assolutamente efficiente, ad esempio mediante l'utilizzo di proxy web.

Il problema fondamentale di ICESHIELD rimane mascherato ed è quello di essere un controllore all'interno dell'ambiente da controllare: non è garantita l'impossibilità per un eventuale codice attaccante di identificare la presenza del sistema di protezione, né viene garantito che l'attaccante possa bypassare la protezione sfruttando vulnerabilità insite nella stessa architettura JavaScript.

Il sistema si basa poi sull'assunto che la macchina d'esecuzione JavaScript rispecchi completamente le specifiche ECMAScript di riferimento: eventuali difformità d'implementazione tra i vari interpreti e/o bug tecnici

specifici potrebbero infatti compromettere l'architettura di base e rendere inefficace o facilmente superabile il blocco di protezione.

Capitolo 3

ScriptShark

Come già anticipato, l'obiettivo di questa tesi è quello di realizzare un sistema completo non solo in grado di analizzare ed identificare le minacce JavaScript, ma anche di proteggere attivamente il browser.

Abbiamo realizzato il sistema ScriptShark con una classica architettura client - server, nel quale il client è rappresentato idealmente da un'estensione per il browser che si desidera proteggere, mentre il server è il punto centrale di analisi e raccolta dati. In questo modo vengono disaccoppiati i due elementi fondamentali del sistema: la parte di analisi ed identificazione delle minacce, che opera sul server, mentre la parte di protezione attiva è applicata al browser mediante l'utilizzo di plug-in. Le due parti del sistema comunicano sfruttando il protocollo standard HyperText Transport Protocol (HTTP): in questo modo è possibile estendere il plug-in del browser anche ad altre architetture o addirittura erogare il servizio di analisi direttamente come servizio Web in modo semplice e diretto.

3.1 L'impianto lato server

Il sistema ScriptShark lato server comprende il sistema di raccolta dati, l'analizzatore di codice ed il sistema di identificazione delle minacce. In

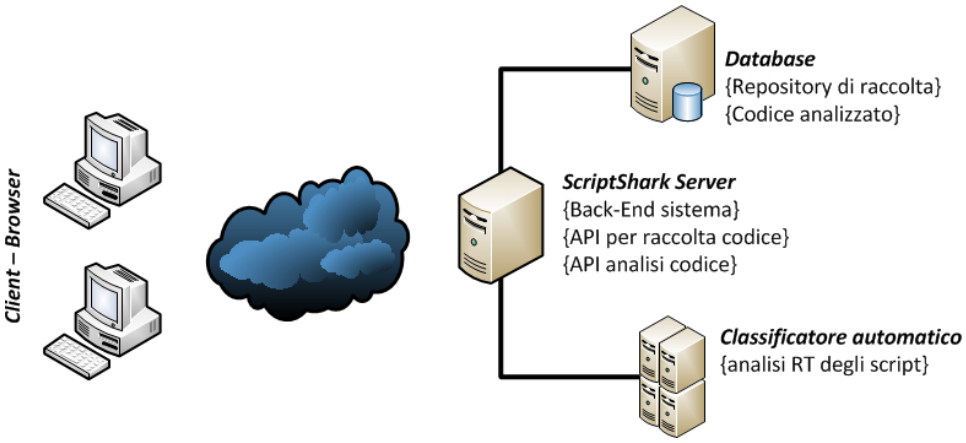

Figura 3.1: Schema completo del sistema ScriptShark

particolare si distinguono tre componenti distinti:

- Server API: è il server Web che risponde alle richieste HTTP inviate dai client del sistema, esponendo quindi le interfacce di comunicazione fondamentali, è basato sull'interprete dinamico PHP.
- Sistema DBMS: concretamente basato su un'istanza MySQL di Oracle, gestisce il repository del codice JavaScript raccolto e analizzato, necessaria inoltre per il funzionamento del sistema in una delle due modalità operative.
- Sistema di classificazione on-demand: è il componente realizzato specificatamente per rendere sostenibile la classificazione in tempo reale del codice sottoposto. Il sistema concreto risulta essere un'applicazione Java standalone con comunicazione basata su socket e implementazione multi-threading, che integra al suo interno l'ambiente di elaborazione Weka. Con questo accorgimento, come verrà mostrato nel Capitolo 5, è possibile ottenere delle performance nettamente superiori all'accesso singolo, per ogni richiesta, al sistema Weka.

3.2 Il plug-in per il browser

Come già introdotto, l'implementazione del sistema per il browser è stata realizzata attraverso un componente plug-in: nel nostro caso specifico, la piattaforma di riferimento è stata Mozilla Firefox, per il quale è stata costruita un'estensione capace di interagire con il debugger JavaScript integrato, al fine di intercettare tutti gli script che vengono creati nel momento in cui sono in procinto di entrare in esecuzione e pilotare la loro eventuale interruzione. La scelta del browser Firefox come piattaforma iniziale di sviluppo è legata alla grande comunità di sviluppo che lo sostiene: è stato infatti possibile accedere ad un vasto archivio di documentazione ed esempi pratici, oltre ad altre estensioni già sviluppate, che hanno velocizzato lo sviluppo.

Il sistema di intercettazione del codice sfrutta due punti d'aggancio del sistema di debugging, come peraltro evidenziato in Figura **??**:

- `onScriptCreated`: qui vengono intercettati i fenomeno di caricamento degli script da parte del motore di rendering del browser; in questo contesto non è però possibile operare sull'esecuzione del codice, nè recuperare alcun dato sulla sua origine. E' però possibile imporre dei punti d'arresto per il debugger sullo script in esame, in particolare è possibile imporne uno sulla cosiddetta *linea zero* dello script: così facendo viene invocato il debugger prima che il codice inizi realmente ad essere eseguito.
  Con questo stratagemma il sistema è in grado di passare in un contesto di analisi più profondo e ricco di dati senza perdere la peculiarità dell'intercettazione del codice prima che esso venga effettivamente eseguito.

- `onExecutionHook`: questo è il gestore eventi per i punti d'arresto applicati al codice in fase di debug; da qui possiamo accedere al con-

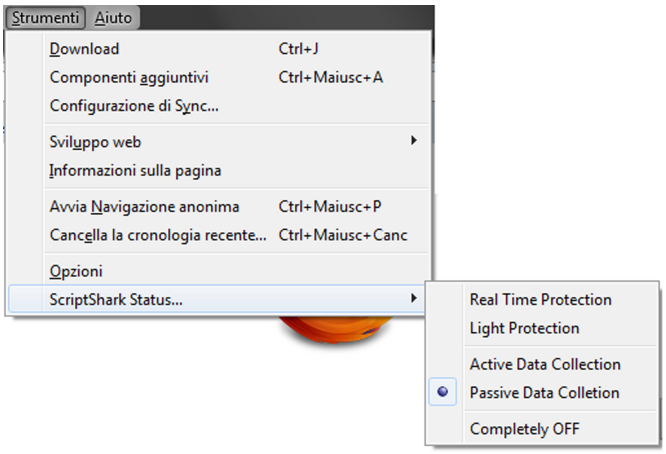

Figura 3.2: Menu di gestione dell'estensione ScriptShark installata in Firefox

testo completo dello script (codice grezzo, URL origine script) ed anche al contesto d'esecuzione dello script (accesso alle variabili JavaScript ed al DOM della pagina caricata). E' inoltre possibile, essendo questo un punto di debug, indicare all'esecutore se è necessario procedere regolarmente nell'esecuzione oppure è opportuno interromperla, esattamente lo scopo per il quale l'estensione è stata sviluppata.

L'utilizzo combinato dei due metodi del debugger JavaScript consente quindi di intercettare attivamente tutto il codice in preparazione per l'esecuzione e, successivamente, di pilotarne l'effettiva esecuzione in base al risultato del nostro processo di analisi.

L'estensione è stata inoltre utilizzata per effettuare la raccolta dati necessaria allo sviluppo del classificatore, descritto in dettaglio nel Capitolo 4. La sua capacità di intercettare il codice è stata infatti sfruttata per realizzare una modalità operativa specifica, nella quale l'estensione opera solamente in raccolta dati, senza interrogare attivamente il sistema di protezione.

Il vantaggio di questa soluzione è duplice: abbiamo infatti realizzato uno strumento di raccolta dati in modo estremamente rapido, basandoci

sul sistema già costruito, ottenendo inoltre l'omogeneità del codice raccolto, in quanto il codice stesso viene raccolto a valle del parsing da parte dell'interprete JavaScript integrato nel browser, semplificando notevolmente la parte di analisi. Inoltre, quando l'estensione opera in modalità di raccolta, senza quindi offrire il servizio di protezione attiva, l'invio dei dati verso il server di raccolta opera in modalità completamente asincrona, evitando quindi di appesantire il processo di rendering ed esecuzione della pagina stessa, minimizzando la possibilità per il sistema di raccolta di essere identificato dal codice malevolo tramite l'analisi del ritardo d'esecuzione. Molti attacchi, infatti, presentano sistemi di analisi del browser e del contesto d'esecuzione volti ad identificare eventuali particolarità, come ad esempio il ritardo durante l'esecuzione di istruzioni semplici o nomi e versioni del browser non corrispondenti alla produzione reale, per rilevare la presenza di analizzatori e sistemi di protezione: in questi casi, il codice può autonomamente decidere di non svilupparsi completamente, impedendo il buon esito della raccolta dati riguardo al tentato attacco.

Il nostro sistema di raccolta opera quindi utilizzando sistema operativo e browser perfettamente standard, salvo la presenza dell'estensione ScriptShark in quest'ultimo: in questo modo si minimizzano le possibilità per il codice malevolo di riconoscere il nostro sistema di raccolta dati e si realizza una raccolta dati estremamente efficace, oltre che realistica, essendo basata su di un browser completo invece che un semplice crawler.

3.2.1 Il risultato semi-statico

L'estensione ScriptShark, come strumento di intercettazione codice JavaScript, raggiunge un risultato tecnologico che, allo stato attuale, è noto solamente in un altro sistema d'analisi (ZOZZLE): il sistema risulta essere in grado di intercettare ed identificare sia il codice incluso all'interno delle pagine HTML renderizzate, oltre a quello collegato da risorse remote, sia il

codice che *dinamicamente* è generato e mandato in esecuzione.

Questa particolarità è legata sia ad alcune possibilità programmatiche esplicitamente esposte dall'ambiente JavaScript, come il metodo `eval(...)`, che manda in esecuzione il codice JavaScript presente nella stringa passata come parametro, ed il costrutto `new Function(...)`, che consente di costruire una vera e propria funzione richiamabile a piacere sempre partendo da codice JavaScript contenuto in una stringa passata per parametro, sia al particolare contesto d'esecuzione, quale è una pagina Web in rendering all'interno del browser: uno script JavaScript può infatti accedere in scrittura al modello della pagina renderizzata (DOM) ed aggiungere ulteriori elementi, contenenti a loro volta del codice JavaScript, che risulta quindi essere generato dinamicamente durante l'esecuzione.

Il sistema globalmente opera quindi con un approccio definibile *semi-statico*.

Il sistema in generale risulta inoltre essere completamente flessibile nei confronti del codice dinamico, in quanto il sistema di intercettazione e raccolta prescinde completamente dalla modalità di creazione del codice, focalizzandosi sull'evento generale di creazione dello script e non sulla modalità con le quali l'evento è stato effettivamente generato. Eventuali nuove tecniche per la creazione di codice durante l'esecuzione risultano quindi già coperte.

3.3 L'analizzatore statico

L'analizzatore rappresenta il componente preliminare del sistema di rilevazione delle minacce. L'analizzatore infatti elabora il codice JavaScript raccolto e ne estrae i dati caratteristici che verranno utilizzati dal classificatore per rilevare il grado di pericolosità del codice esaminato.

Come già anticipato, l'analisi avviene in via completamente statica: l'i-

dea è infatti quella di verificare se è possibile riconoscere la pericolosità di un frammento di codice analizzandone staticamente nel codice gli elementi propri del linguaggio (attributi e metodi) capaci di costituire l'architettura di una possibile azione malevola.

Oltre al codice grezzo sono analizzati anche i dati di contesto disponibili con ogni script, quali:

- URL pagina visitata
- URL origine script
- Tipologia/denominazione script

Le feature estratte dall'analizzatore sono quindi passate al classificatore, che in base all'addestramento ricevuto, può fornire la propria stima del grado di pericolosità del codice esaminato. Il dettaglio del modello utilizzato nel classificatore e della relativa parte di addestramento supervisionato sono descritti nel Capitolo 4.

3.3.1 Feature analizzate sul codice

Sono state identificate 8 feature d'analisi, ognuna mirata ad indentificare nel codice una particolare propensione all'esecuzione di un'attività potenzialmente malevola; per ogni feature sono presenti in numero variabile delle chiavi di ricerca che vengono verificate testualmente sul codice.

Le feature categoriche analizzate e la loro organizzazione sono state suggerite dai lavori precedenti, quali WEPAWET e ZOZZLE, e sono state completate con la ricerca specifica, il più possibile esaustiva, di tutte le chiavi testuali, che verranno effettivamente cercate all'interno del codice, inerenti quelle particolari caratteristiche.

- Redirezioni (4 elementi)

  Vengono identificati nel codice tutte le potenziali operazioni che por-

tano alla redirezione del browser su di un'altra pagina. Come ad esempio l'assegnamento alla proprietà `location.href=...`.

- ActiveX e plug-in (1 elemento)
  Vengono identificate le creazioni di ActiveX e plug-in in generale; è attraverso questi componenti che tipicamente un attacco JavaScript riesce ad iniettare ed eseguire dello shellcode.
  Viene identificata la presenza del costrutto `new ActiveXObject(...)`.

- Manipolazione stringhe (7 elementi)
  Si evidenziano le operazioni di manipolazione stringhe, in particolare quelle che possono allocare grandi aree di memoria; queste operazioni sono frequentemente utilizzate per costruire in memoria strutture estese e compatte atte a contenere il codice malevolo negli attacchi drive-by e code injection. Come ad esempio `concat()`, `slice()`, `toUpperCase()`.

- Codice dinamico (3 elementi)
  Si verifica se all'interno del codice è utilizzato il sistema di esecuzione dinamica proprio di JavaScript, quindi i metodi `eval(...)` e `setTimeout(...)`, oltre al costrutto `new Function(...)`.

- Scritture DOM (5 elementi)
  Si cercano nel codice evidenze di operazioni di scrittura sul DOM, siano esse di pura aggiunta (iniezione di nuovi contenuti/tag all'interno della pagina) che di sovrascrittura. Come ad esempio `innerHTML=...` e `appendChild(...)`.

- Elementi sospetti (12 elementi)
  Si identificano nel codice eventuali elementi che per loro natura possono essere sospetti, come `<script>` ed `<embed>`, e le stringhe rela-

tive ad alcuni tipi di plug-in noti per essere usati spesso come veicolo di attacco, come `Shell.Application`.

- Offuscamento (7 elementi)
  Si cercano nel codice evidenze sull'utilizzo di funzioni e codifiche capaci di nascondere del codice all'interno di stringhe regolari, come ad esempio la decodifica dal charset Unicode o dal sistema esadecimale, come `fromCharCode(...)` e `unescape(...)`.

- Identificazione (1 elemento)
  Si verifica se lo script accede all'oggetto `navigator`, ovvero se il codice cerca in qualche modo di identificare il browser/sistema che lo sta effettivamente eseguendo.

Per ognuna di queste feature viene quindi registrato il numero di occorrenze dei suoi elementi all'interno del codice, consentendo di avere un'indicazione su quanto frequenti sono le specifiche operazioni di quella categoria in quel particolare segmento di codice. In questo modo abbiamo estratto in modo sintetico un'indicazione generale della misura in cui il segmento di codice in analisi contiene attività relative a quella particolare categoria.

3.3.2 Feature analizzate sul contesto

Oltre alle feature specificatamente ricercate nel codice, il sistema valuta anche i dati di contesto associati al codice in esame.

In particolare vengono valutati:

- Lunghezza del frammento codice
  Viene calcolata la lunghezza in caratteri del frammento di codice.

- Tipo di schema HTTP utilizzato per pagina e origine script
  Si confrontano gli schemi d'accesso utilizzati per accedere alla pagina ed all'origine script, per verificare se la discrepanza tra un accesso

sicuro ed uno non sicuro, per risorsa, possono diventare identificatori per la pericolosità dello script.
Per ogni sample vengono estratte le informazioni sulla cifratura del canale (HTTP - HTTPS) di pagina e origine script, che vengono poi combinate per generare l'indicazione della caratteristica.

- Dominio d'origine per pagina e script
  Si confrontano i domini d'origine per pagina e script, al fine di verificare se lo script proviene dallo stesso fornitore della pagina oppure viene recuperato da terze parti.

- Tipologia/denominazione script
  Lo strumento di raccolta dei sample fornisce un'indicazione sul nome del frammento codice, attraverso il quale è possibile classificare lo script in una tra tre tipologie distinte:

  - script principale, ovvero codice che risulta incluso direttamente nella struttura HTML della pagina esaminata oppure codice originato da terze parti
  - script di funzione dichiarata, ovvero codice che rappresenta il corpo di una funzione ben definita dal codice, dotata quindi di un riferimento testuale completo
  - script anonimo, ovvero un frammento di codice derivante dall'esecuzione dinamica (`eval`,`new Function`) oppure corpo di una funzione dichiarata in-line e quindi priva di definizione e riferimento testuale.

3.4 Il sistema di protezione

La protezione offerta dal sistema ScriptShark può operare in due differenti modalità:

- Protezione base: in questa modalità il sistema opera un controllo sul grado di pericolosità o meno del singolo script intercettato interrogando una base dati centralizzata, contenente codice già analizzato, e prosegue o interrompe l'esecuzione in base al responso ottenuto.
  Nel caso in cui lo script non risulti già noto, di base è assunto come lecito e si prosegue l'esecuzione, in modo da non penalizzare il sistema nelle prime fase di database non completo.

- Protezione real-time: in questa modalità, nel caso in cui lo script intercettato non sia già noto al sistema, si procede ad una analisi in tempo reale mediante l'uso del classificatore, la cui costruzione è dettagliata nel capitolo successivo.

L'estensione per il browser valuta ogni script di codice JavaScript intercettato, chiedendo attivamente al server centrale una valutazione sul grado di pericolosità dello stesso, in base alla quale verrà deciso la prosecuzione o l'interruzione dell'esecuzione. Il funzionamento dell'intero sistema è quindi fortemente sincrono ed eventuali ritardi di comunicazione inficiano in modo diretto sull'esperienza utente di navigazione: questa caratteristica non è stata trascurata e l'estensione stessa è stata oggetto di un'attenta analisi volta a studiare varie soluzioni e relativi benefici volti a minimizzare l'impatto sull'esperienza di navigazione globale.

3.4.1 Compressione dati

Si è cercato di valutare l'impatto che un semplice sistema di compressione dati potesse avere sul processo di comunicazione tra l'estensione del browser ed il server. Il sistema scelto è stato un classico LZMA, algoritmo di compressione di semplice implementazione e disponibile in forma interoperante per vari linguaggi, tra cui proprio JavaScript (per il client) e PHP (per il server).

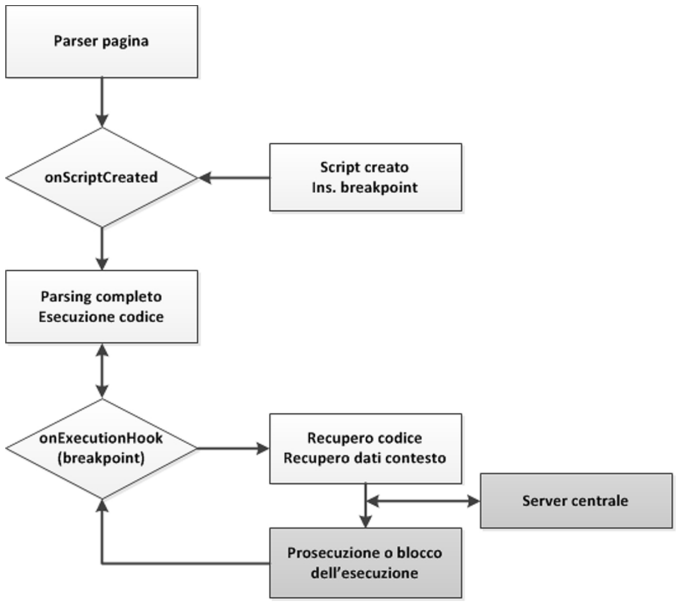

Figura 3.3: Schema di funzionamento dell'estensione ScriptShark in protezione

Gli scarsi risultati ottenuti dal sistema, esposti in dettaglio nel Capitolo 5, hanno decretato la rimozione di questa caratteristica implementativa dal sistema in sviluppo.

3.4.2 Semplificazione dati

Pur non potendo applicare dei sistemi di compressione dati per limitare la quantità di dati in transito sulla rete, si è cercato di ridurre al minimo i dati effettivamente scambiati tra l'estensione e la controparte server.

Per la modalità di analisi real-time non si è potuto intervenire ulteriormente, in quanto tutti i dati in transito (URL pagina, URL script, tipologia script, codice grezzo) sono effettivamente necessari ed indispensabili per l'analisi.

Per la modalità di analisi base invece si è potuto ottimizzare fortemente i dati in transito. In questa modalità infatti non è necessario che il client comunichi al server tutti i dati raccolti, lo scopo dell'interrogazione è in-

fatti solo quello di identificare se quello specifico script raccolto è già noto al sistema, ovvero se è già presente all'interno del database, e, nel caso, se è riconosciuto come codice malevolo oppure no. Con queste premesse è evidente come la trasmissione dell'intero set di dati raccolti, composto da codice dello script e dati di contesto, risulta essere inutilmente ampio e privo di reale utilità. Si è quindi scelto di ridurre il set di dati comunicati ad un solo codice di hash, calcolato sul solo codice grezzo dello script raccolto: in questo modo il client può indicare di quale script vuole il responso in modo semplice e compatto. Lato server inoltre, la ricerca di una semplice stringa di lunghezza fissa all'interno di una tabella, diventa un compito estremamente veloce e ottimizzato, alla luce anche del fatto che su tale campo di ricerca viene mantenuto un indice apposito per ottenere il massimo delle prestazioni dal Database Management System (DBMS).

3.4.3 Cache risultati

Nel mondo della programmazione generale vige l'assunto della "località spaziale e temporale dei dati", ovvero che se un'istruzione accede ad una determinata locazione di memoria è altamente probabile che nel breve termine si verificherà un accesso alla medesima locazione (località temporale) o che verrà richiesto l'accesso ad una locazione di memoria vicina a quella appena acceduta (località spaziale).

L'idea che rappresenta il fondamento dell'assunto appena descritto è facilmente applicabile anche nel contesto della navigazione Web: se un utente ha avuto accesso ad una determinata pagina, è molto probabile e plausibile che nel breve termine esso cercherà di accedere ad una pagina del medesimo dominio di risorse e che parte dei dati già caricati con la prima pagina vengano riutilizzati anche dalla nuova. E' molto importante quindi far sì che il sistema sia in grado di mantenere una memoria a breve termine del codice analizzato, in modo tale da non appesantire il server con inu-

tili richieste multiple per lo stesso script e velocizzare così la navigazione dell'utente.

E' stato quindi introdotto un sistema di cache per i risultati ottenuti dal sistema d'analisi, indicizzato usando il codice hash dello script stesso. Il sistema prevede inoltre un meccanismo automatico di pulizia della cache basata su scadenza del contenuto dopo l'ultimo accesso rilevato.

Nel Capitolo 5 verrà mostrato l'incremento prestazionale legato all'implementazione di questa caratteristica.

3.4.4 Riduzione timeout chiamate sincrone dell'estensione

Il problema principale dell'estensione è la necessità di affidarsi a chiamate sincrone verso il server per verificare il grado di pericolosità del codice. Questo, avvenendo durante l'esecuzione del codice JavaScript da parte del browser, provoca un rallentamento notevole e non indifferente nell'esperienza di navigazione dell'utente.

Se da un lato si determina la necessità di avere come impianto server un sistema molto performante che consenta di rispondere alle richieste impiegando il minimo tempo possibile, dall'altro è importante che anche l'estensione sia in grado di interrompere la chiamata nel momento in cui il tempo di risposta stia diventando eccessivamente lungo.

Questa modifica sull'estensione garantirebbe inoltre due benefici ulteriori:

- Fail-off automatico in caso di irraggiungibilità del server: se il server dovesse diventare irraggiungibile durante l'uso, per un problema dell'impianto lato server o per un problema della connettività lato utente, l'estensione potrebbe identificare il problema e disattivarsi autonomamente, previa segnalazione all'utente, per non inficiare ulteriormente sull'esperienza di navigazione, evitando quindi di rimanere attiva pur non generando beneficio di alcun tipo.

- Load-Balancing automatico: se il tempo di risposta del server rilevato dall'estensione dovesse salire oltre una certa soglia per più di un determinato intervallo temporale, l'estensione potrebbe autonomamente ipotizzare un potenziale sovraccarico del sistema, procedendo alla sospensione del servizio in modo tale da consentire all'impianto server di rientrare naturalmente in una condizione di carico accettabile.

Pur essendo questa caratteristica particolarmente importante e fondamentale per il buon miglioramento dell'estensione, non è stato possibile implementare nulla di tutto questo in quanto il target della nostra estensione, il browser Mozilla Firefox, non supporta, alla versione attuale (v10), la funzionalità di timeout per le richieste remote generate in modalità sincrona. Il suo supporto è previsto a partire dalla futura versione 12 e pertanto questa miglioria sarà introdotta solamente in futuro durante l'evoluzione dell'estensione.

Capitolo 4

Valutazione del sistema di identificazione

La sezione del sistema ScriptShark dedicata all'identificazione del codice è stata realizzata mediante un classificatore in grado di esaminare le feature, estratte dall'analizzatore, al fine di rilevare il grado di pericolosità del codice.

L'impossibilità di conoscere a priori eventuali legami esistenti tra le caratteristiche del codice ed il suo grado di pericolosità hanno determinato la necessità di costruire un classificatore mediante l'utilizzo di tecniche di apprendimento supervisionato.

4.1 Raccolta dati e composizione del dataset

Per addestrare il classificatore abbiamo innanzitutto dovuto raccogliere sample individuali di codice JavaScript grezzo in quantità notevoli. Purtroppo, non esistono, disponibili pubblicamente, dei database di codice JavaScript grezzo che contengano contemporaneamente sample di codice valido e sample di codice malevolo etichettati puntualmente. Il nostro obiettivo è invece quello di raccogliere sia sample malevoli, sia sample leciti, con

la granularità del singolo frammento di codice, in modo da poter, su questa base, sia costruire un sistema d'identificazione automatico, sia valutare l'efficacia dello stesso in un contesto realistico.

Con il plug-in ScriptShark per il browser Firefox, descritto in dettaglio nel precedente capitolo, abbiamo costruito un dataset specifico per lo sviluppo del classificatore, attraverso una navigazione automatizzata di un elenco di URL di riferimento.

A tale scopo, abbiamo realizzato uno script che effettua la navigazione automatizzata pilotando il browser Firefox, presente a bordo di una macchina virtuale, al fine di lanciare la navigazione non presidiata di un elenco specifico di URL. Come già anticipato nel Capitolo 3, il sistema di raccolta utilizza sistema operativo e browser regolari e non specificatamente modificati, fatta eccezione la presenza dell'estensione nel browser: questo minimizza la possibilità per il codice malevolo di identificare il nostro sistema d'analisi ed impedire, di conseguenza, il regolare caricamento, e la successiva raccolta, del codice malevolo.

Il sistema è stato eseguito su di un set ridotto della Top Ranking Alexa, con lo scopo di raccogliere sample di codice lecito, e successivamente su di un elenco di risorse recuperato dal servizio malwaredomainlist.com (MDL), che mantiene un database di URL segnalati come infetti.

Tra gli URL estratti dal servizio MDL, un numero molto elevato è risultato essere offline o in generale non più raggiungibile, perché risalente a vecchie segnalazioni, ormai risolte. Globalmente abbiamo quindi effettivamente visitato 35.203 siti appartenenti alla Top Ranking Alexa e 7.456 URL provenienti dal servizio MDL, per un totale di 5.078.595 "sample", dove per sample si intende un singolo frammento di codice JavaScript carico di tutte le informazioni di contesto relative.

Dal dataset sono successivamente stati estratti i soli sample *unici*, usando come chiave di discriminazione un codice di hash MD5 calcolato sull'in-

terno codice grezzo raccolto. In questo modo si sono isolati i frammenti di codice che risultavano essere realmente diversi tra loro, prendendo in considerazione per la composizione del dataset finale solamente sample unici e velocizzando la fase di analisi, evitando quella di frammenti già esaminati.

In definitiva il dataset comprende 489.770 sample unici.

4.1.1 Etichettatura del dataset

Per addestrare il sistema di classificazione sfruttando un metodo di apprendimento supervisionato, è necessario che i sample presenti nel dataset siano *etichettati*, ovvero che ad ogni sample venga attribuito a priori un grado di pericolosità.

Ogni sample è stato quindi etichettato in base sia alla provenienza originale del sample, se proviene dal crawling della Top Ranking Alexa oppure dal crawling di MDL, sia con la verifica puntuale in linea con il sistema Google Safebrowsing. Il sistema di Google ha confermato per tutti gli URL provenienti da Alexa l'affidabilità del codice, mentre per alcuni URL provenienti da MDL ha annullato la segnalazione di pericolosità, presumibilmente per minacce già corrette e/o rimosse dalla risorsa.
I sample per cui non corrispondeva la valutazione di pericolosità tra il sistema SafeBrowsing e l'effettiva provenienza sono stati scartati per evitare ogni tipo di ambiguità ed indecisione nell'etichettatura iniziale del dataset.

4.1.2 Problema nell'etichettatura

Il dataset così costruito è risultato essere però affetto da un errore di base, legato proprio alla fase di etichettatura del codice malevolo. Se è infatti possibile affermare correttamente che se un URL non risulta essere segnalato come malevolo, allora tutto il codice in esso contenuto sarà necessariamente non malevolo, l'affermazione contraria non gode dello stesso livello di certezza.

Il codice malevolo infatti non è tipicamente ospitato completamente su pagine dedicate allo scopo ma risulta essere spesso iniettato all'interno di siti e risorse perfettamente regolari: in questo modo è più semplice per il malintenzionato colpire un elevato numero di sistemi senza la necessità di forzarli a raggiungere un particolare URL. Gli URL che vengono effettivamente segnalati tramite i servizi quali MDL sono per l'appunto risorse di carattere generale che risultano infette da codice malevolo, confermando ancor di più l'importanza di questo aspetto.

Se quindi un URL è segnalato come malevolo è altamente probabile che solo una minima parte del codice presente su quella risorsa rappresenti effettivamente il codice pericoloso, mentre la maggior parte del codice risulterà essere perfettamente lecita, in quanto costituente la vera struttura della risorsa originale.
Questa caratteristica provoca l'errata etichettatura di buona parte dei sample presenti nel dataset e segnalati come malevoli: tali sample potrebbero essere infatti codice perfettamente lecito, costituendo un pericolo per l'efficacia finale di qualsiasi classificatore addestrato sul dataset.

Non è però possibile risolvere questa anomalia. Tutti i sistemi disponibili di analisi minacce in JavaScript operano infatti a livello di singolo URL e non esiste un servizio che fornisca dettagli sul singolo frammento di codice. Pertanto il dataset così etichettato è il migliore disponibile con il quale procedere e non esistono strumenti capaci di migliorare sensibilmente l'affidabilità dell'etichettatura iniziale.

Questo difetto del dataset non è comunque stato ignorato, anzi, è stato oggetto di un'attenta valutazione volta a minimizzarne il più possibile gli effetti, rappresentata da una fase di sanitizzazione pre-analisi descritta nella sezione successiva.

Sample Benigni	Sample Malevoli	Totale
459.122	30.648	489.770
(93,7 %)	(6,3 %)	(100,0 %)

Tabella 4.1: Composizione del dataset originale

4.2 Sanitizzazione iniziale del dataset

Come accennato nella sezione precedente, il dataset risulta affetto da un potenziale errore di etichettatura, che affligge esclusivamente i sample segnalati come malevoli.
Questo problema potrebbe portare ad una valutazione errata del classificatore, sia in fase di addestramento, in quanto vengono forniti al classificatore come malevoli sample in realtà leciti, sia in fase di valutazione, in quanto gli elementi classificati e riconosciuti come falsi negativi potrebbero in realtà essere l'indicazione del buon funzionamento del classificatore stesso, fornendo quindi risultati sintetici non attendibili.

Si è quindi proceduto ad una sanitizzazione inizale del dataset, al fine di rimuovere dall'insieme dei sample malevoli quei frammenti di codice che molto probabilmente sono in realtà frammenti corretti. La pre-analisi ha coinvolto un classificatore di tipo SVM a singola classe, addestrato su un sottoinsieme dei sample buoni (58.727 sample per l'esattezza). Non è stato possibile costruire il classificatore sull'interno dataset di sample buoni per problemi di performance e limiti intrinseci dello strumento Weka.

L'obiettivo perseguito è stato quindi quello di ottenere un classificatore capace di identificare se un frammento di codice è, con buona confidenza, lecito oppure no. Il classificatore così realizzato ha consentito di ri-etichettare ben 10.699 sample, classificandoli come benigni. Una verifica manuale su 50 sample casualmente estratti ha confermato empiricamente la validità dell'operazione, evidenziando come tutti i sample analizzati

Sample Benigni	Sample Malevoli	Totale
469.821	19.949	489.770
(95,9 %)	(4,1 %)	(100,0 %)
URL Benigni	URL Malevoli	Totale
27.089	6.632	33.721
(80,3 %)	(19,7 %)	(100,0 %)

Tabella 4.2: Composizione del dataset sanitizzato

risultassero essere frammenti di codice non malevolo.

In Tabella 4.2 è possibile vedere la composizione finale del dataset sanitizzato utilizzato nella succesiva fase di creazione del classificatore automatico.

4.3 Comparativa modelli e risultati

Non conoscendo a priori nulla riguardo la tipologia e la complessità del problema che si sta affrontando, il dataset è stato utilizzato per costruire vari modelli di classificatori automatici, al fine di identificare quello con le migliori performance complessive. Weka si è rivelato uno strumento importate sotto questo aspetto, perché ha consentito di mettere alla prova un insieme variegato di modelli, diversi tra loro, in breve tempo ed ottenendo risultati comparabili.

Ogni modello testato è stato costruito usando l'intero dataset con la tecnica della cross-validazione a 10 gruppi e valutato comparando i tassi di falsi positivi, ovvero il numero di sample leciti identificati come malevoli, e falsi negativi, ovvero il numero di sample malevoli identificati come leciti, ottenuti da ogni modello.

Come è evidente dalla Tabella 4.3, il modello ad albero J48 ha dimostrato di avere i migliori risultati rispetto ai concorrenti del lotto analizzato, ed

Modello	False Negative (FN)	False Positive (FP)
J48 Tree	34.2 %	0.6 %
Bayesian Net	63.8 %	2.1 %
Naive Bayesian	10.8 %	45.2 %

Tabella 4.3: Modelli esaminati a confronto

Modello	FN	FP
J48 Tree	34,2 %	0,6 %

Tabella 4.4: Risultati del modello J48 sul dataset sanitizzato

è quindi stato scelto come modello definitivo.

4.4 Analisi del modello J48 e raffinamento successivo

Come già presentato in Tabella 4.3, il modello J48 è stato in grado di ottenere questi risultati sintetici:

A fronte di un ottimo risultato in termini di falsi positivi, casi in cui il sistema compie un errore identificando come malevolo un sample perfettamente lecito, il risultato in termini di falsi negativi, casi in cui il sistema non riesce ad indentificare una minaccia nota, rappresenta invece un risultato non ottimale, sotto le aspettative, per un sistema che si prefigge lo scopo di identificare attivamente il codice malevolo.

L'anomalia, purtroppo, è ancora riconducibile al problema di etichettatura iniziale che affligge il dataset, descritto precedentemente. Sicuramente l'operazione di sanitizzazione iniziale ha contribuito a migliorare il dataset, riducendo sensibilmente la quantità di sample non malevoli etichettati erroneamente, ma l'anomalia permane: una verifica manuale su un set di 50 sample estratti casualmente dall'elenco dei falsi negativi ho confermato che tali sample risultavano essere non malevoli e, pertanto, etichettati

Sample Benigni	Sample Malevoli	Totale
473.470	16.300	489.770
(96,7 %)	(3,3 %)	(100,0 %)

Tabella 4.5: Composizione del dataset ulteriormanete raffinato

erroneamente.

Questo problema relativo al dataset non è, peraltro, risolvibile in modo definitivo e rigoroso: come infatti già esposto precedentemente, non esiste al momento un altro sistema di analisi del codice JavaScript in grado di fornire risultati a livello di singolo frammento codice e, pertanto, non è possibile correggere in modo rigoroso e definitivo l'etichettatura dei sample malevoli.

Il dataset è stato pertanto sottoposto ad un secondo raffinamento, basato sull'idea che, se il sistema ha correttamente identificato per una determinata risorsa URL un set di sample malevoli, i restanti sample provenienti dalla medesima risorsa, che risultano etichettati come malevoli in virtù della comune origine, ed identificati come non malevoli dal classificatore, rappresentano realmente, con buona probabilità, sample non malevoli. Con questo assunto si è raffinato il dataset aggiornando l'etichetta dei sample che corrispondevano alle condizioni appena descritte: in particolare 3.649 sample sono stati oggetto dell'aggiornamento, portando il dataset raffinato ad avere la composizione riportata in Tabella 4.5. L'operazione è stata empiricamente validata con l'analisi manuale di 300 sample estratti casualmente, confermando in tutti i casi la non pericolosità del codice e, conseguentemente, l'errore di etichettatura iniziale.

Su questo nuovo dataset è stato ricostruito il modello J48, ottenendo i risultati finali indicati in Tabella 4.6.

Il raffinamento ha consentito di rimuovere un numero considerevole di errori di identificazione, rietichettando correttamente i sample non

Modello	FN	FP
J48 Tree	19,7 %	0,6 %

Tabella 4.6: Risultati del modello J48 sul dataset ulteriormente raffinato

malevoli che risultavano falsi negativi nella precedente analisi.

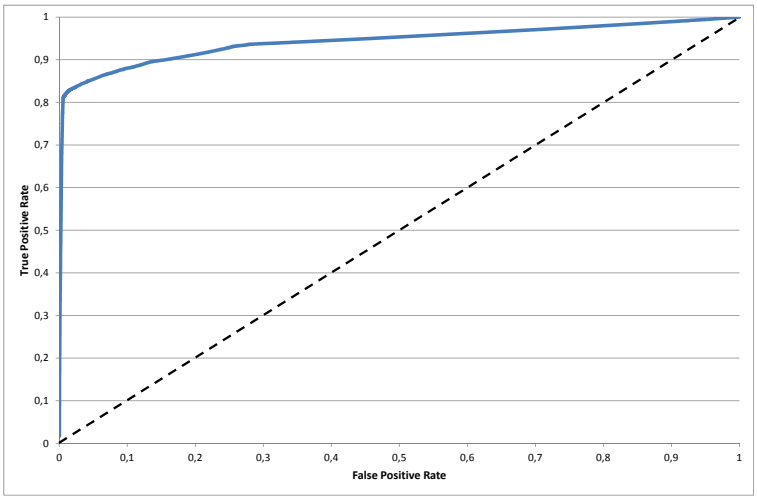

Figura 4.1: Curva ROC del modello J48 sul modello raffinato

Alla luce delle nuove valutazioni, si può affermare che il classificatore ottiene, globalmente, dei buoni risultati: il modello ha dimostrato una discreta capacità di identificazione del codice malevolo, mantenendo al contempo un bassissimo tasso di falsi positivi. Questo candida il modello ad essere utilizzato attivamente in un potenziale sistema di protezione reale, come quello prototipale descritto nel capitolo seguente.

	FN (URL)	FP (URL)
Identificatore ScriptShark	21,7 %	6,2 %

Tabella 4.7: Risultati di identificazione a livello URL

4.5 Copertura URL

Per approfondire lo studio dei risultati ottenuti dal classificatore, abbiamo espanso la granularità dell'analisi portandola dal singolo sample, il frammento di codice, fin qui oggetto della valutazione del classificatore, al livello degli URL.

La ragione principale di questa ulteriore fase d'analisi è legata allo scopo finale del sistema, ovvero identificare singoli frammenti di codice malevolo eventualmente presenti su di una risorsa Web: possiamo infatti affermare che il sistema è realisticamente funzionante nel momento in cui per ognuno degli URL visitati che risultavano segnalati come malevoli è stato identificato almeno un sample malevolo tra quelli recuperati da quella specifica risorsa.

Si è quindi proceduto all'estrazione degli URL d'origine di tutti i sample identificati come True Positive (TP) per confrontarli con l'elenco originale completo di tutti gli URL segnalati malevoli visitati durante la costruzione del dataset. Uguale approccio è stato utilizzato per verificare la percentuale di risorse lecite all'interno delle quali è stata identificata la presenza di uno script malevolo, estraendo gli URL relativi ai sample identificati come FP.

I risultati ottenuti dal nostro classificatore finale a livello URL sono riportati in Tabella 4.7.

4.5.1 Analisi falsi negativi

Per confermare ulteriormente la validità dell'identificatore, abbiamo analizzato con lo strumento WEPAWET gli URL delle risorse relative ai sample

	%	Responso Wepawet
1.176	93,48 %	Benign
24	1,91 %	Suspicious
1	0,08 %	Malicious
57	4,53 %	Invalid URL
1.258	100,00 %	Totale

Tabella 4.8: Risultati del sistema WEPAWET sul set di URL estratti dai falsi negativi

identificati come falsi negativi dal classificatore.

In Tabella 4.8 sono riportati i risultati ottenuti sul totale di 1.258 URL estratti dai 2.909 sample segnalati come falsi negativi: la maggior parte delle risorse risultano confermate come non malevole e solo una piccola parte risulta essere sospetta o maligna. Questo evidenzia la validità del nostro identificatore, confermando che il numero elevato di falsi negativi riscontrati è dovuto principalmente all'errore di etichettatura descritto all'inizio del capitolo e non ad una oggettiva incapacità del nostro classificatore di riconoscerli correttamente.

Capitolo 5

Prestazioni del sistema di protezione

Dopo aver valutato la capacità di identificazione di minacce JavaScript da parte del nostro classificatore, abbiamo valutato anche le prestazioni del sistema di protezione nella sua interezza, con l'obiettivo di verificare in quale misura il sistema ScriptShark potesse influenzare l'esperienza di navigazione.

5.1 Performance del server

Per la parte server dell'impianto, oltre ad una valutazione globale sulla velocità di elaborazione delle richieste inviate alle API, abbiamo valutato separatamente le prestazioni del sistema di classificazione on-demand, per confermare la necessità dello sviluppo di un'applicazione dedicata a tale compito.

5.1.1 Performance del classificatore on-demand

Come già esposto nel Capitolo 3, per classificare in tempo reale il codice è stato necessario sviluppare un piccolo applicativo standalone, capace di

Chiamata singola	~550 ms	
Weka-Server	5 ms	- 99 %

Tabella 5.1: Performance del sistema di classificazione on-demand dedicato

API	Risposta
Prot. Base	39 ms
Prot. Real-Time	82 ms

Tabella 5.2: Tempi di risposta delle API di protezione, media su 1000 richieste

integrare il sistema Weka, consentendo comunicazione basata su socket e, sopratutto, multi-thread.

Questo, rispetto ad una chiamata diretta e puntuale del classificatore per ogni richiesta, consente di risparmiare l'overhead del caricamento in memoria del sistema di classificazione e della relativa parametrizzazione legata al modello costruito sui dati, che avviene così solo in fase di start-up dell'applicazione server e non individualmente per ogni richiesta di analisi.

Complessivamente quindi, il sistema di classificazione on-demand, ottiene le performance presentate nella Tabella 5.1.

5.1.2 Performance server API

Fondamentali per il buon sostentamento del sistema, risultano essere le performance del server, intese come velocità di risposta alle richieste fatte dal client attraverso le API esposte. Essendo un sistema client-server fortemente sincrono, l'esecuzione delle procedure deve risultare il più possibile veloce, al fine di rendere trasparente e poco invadente il sistema di protezione nell'esperienza globale di navigazione.

In Tabella 5.2 sono riportati i valori medi rilevati per il completamento di una singola richiesta, a fronte di un test complessivo svolto con 1000 richieste consecutive.

Minimo	Media	q_{90}	Massimo	# Sample
129 B	4,82 KB	6,27 KB	1,36 MB	2.795

Tabella 5.3: Payload utilizzati per valutare il sistema di compressione

5.2 Performance dell'estensione Firefox

In relazione agli specifici accorgimenti introdotti nell'estensione per il browser Firefox del sistema ScriptShark, abbiamo analizzato puntualmente i risultati ottenuti dalle tecniche di compressione e di cache locale dei risultati, in modo da confermarne, nel primo caso, l'inefficacia mentre, nel secondo caso, l'effettiva utilità.

5.2.1 Compressione dati

Con questi test abbiamo valutato l'efficacia del sistema di compressione dati esaminato durante l'ottimizzazione dell'estensione per il funzionamento in protezione. I test sono stati eseguiti su un set di payload, ricavati dal vivo su un set di 5 siti di riferimento, con le caratteristiche sintetiche riportate in Tabella 5.3. I risultati ottenuti dal sistema di compressione non sono stati in alcun modo soddisfacenti: si è verificato che i payload più leggeri venivano ingranditi notevolmente (fino al 120 %) e solo i payload superiori a 6KB venivano effettivamente compressi fino al 30 %.

Considerando la distribuzione della dimensione globale dei dati in transito, dalla quale si evince che il 90 % circa dei dati è al di sotto della soglia di buon funzionamento del sistema, solo una limitata parte degli script usufruirebbe realmente di un beneficio tangibile.

Il sistema è stato quindi abbandonato, senza procedere ulteriormente con le analisi relative all'overhead computazionale necessario per il mantenimento del sistema di compressione. Ulteriori sviluppi potranno con-

#	Prima apertura	Aperture successive	
1	8354 ms	1025 ms	- 88 %
2	32487 ms	7844 ms	- 76 %
3	25903 ms	7160 ms	- 72 %
4	18454 ms	8642 ms	- 53 %

Tabella 5.4: Valutazione efficacia del sistema di cache interno

centrarsi sulla ricerca di un algoritmo di compressione efficace per questa particolare tipologia di dati.

5.2.2 Cache risultati

Abbiamo analizzato l'impatto prestazionale che deriva dal sistema di cache interno implementato nell'estensione.

In Tabella 5.4 è possibile vedere i risultati di questa analisi: mentre il primo caricamento risulta essere nettamente più lento, i successivi raggiungono tempi decisamente inferiori, grazie alla riduzione del numero di richieste remote effettuate. Dalla tabella si evince inoltre che l'incremento prestazionale non è costante ma varia in base alla pagina specifica, in particolare alla sua struttura interna: più complessa risulterà essere la parte JavaScript, meno il sistema riuscirà ad essere incisivo.

5.3 Performance del sistema complessivo

Oltre alle valutazioni prestazionali sui singoli componenti, abbiamo inoltre valutato globalmente le prestazioni del sistema, cercando di ottenere dei dati capaci di mostrare l'effettivo impatto sull'esperienza utente finale. L'utilizzo del sistema ScriptShark comporta, come prevedibile, una sensibile diminuzione di performance del browser nel suo complesso.

	Punteggio	
Browser puro	365,49	
Debugger JS Attivo	215,18	- 41 %
Raccolta dati	197,11	- 46 %
Prot. base	197,07	- 46 %
Prot. real-time	196,81	- 46 %

Tabella 5.5: Performance JavaScript misurate con la suite Dromaeo

Il sistema in particolare soffre soprattutto a causa del debugger JavaScript: la sua sola attivazione (intesa come attivazione priva di qualsivoglia strumento di analisi effettivamente connesso) comporta un crollo delle prestazioni pure dell'esecutore JavaScript del 40 % circa. L'estensione qui realizzata non si sottrae al medesimo calo prestazionale, in quanto sviluppata sfruttando appunto il debugger JavaScript già disponibile all'interno del contesto del browser Firefox.

La Tabella 5.5 mostra per l'appunto le prestazioni assolute ottenute dal browser in varie condizioni con un test di benchmarking JavaScript Dromaeo di Mozilla (`http://dromaeo.com/`), una suite di test JavaScript open source che comprende inoltre una parte dei test-set del sistema SunSpider (`http://webkit.org/perf/sunspider-0.9/sunspider.html`). È importante notare come in tutte e tre le modalità operative dell'estensione che utilizzano attivamente la parte di debug JavaScript le performance non calano e rimangono sostanzialmente livellate: questo indica chiaramente che l'onere d'esecuzione della parte peculiare del sistema, ovvero quella relativa all'analisi del codice tramite il server centrale, sia effettivamente di basso impatto, mentre la maggior parte del calo prestazionale è legato all'utilizzo del debugger JavaScript integrato in Firefox.

Il calo prestazionale si configura, in definitiva, come un problema di tipo esclusivamente implementativo: un possibile sviluppo potrebbe es-

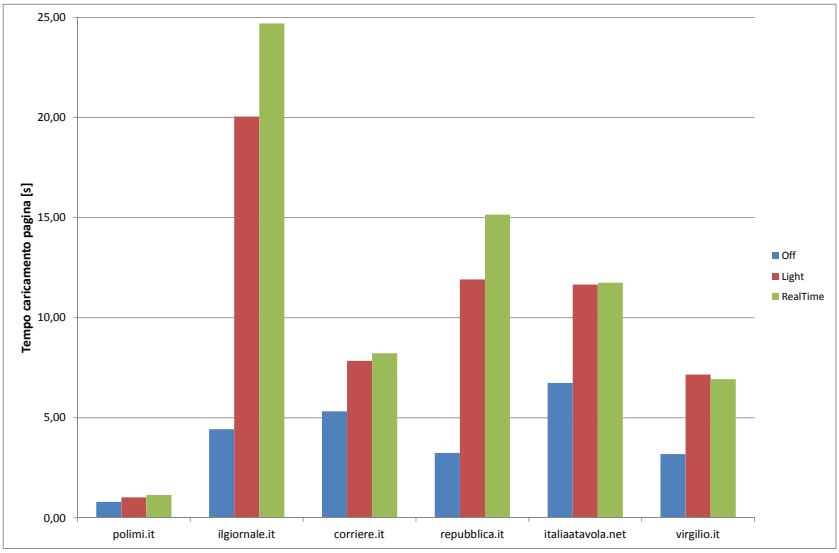

Figura 5.1: Performance del sistema ScriptShark - Caricamento pagine

sere infatti quello di realizzare un sistema di debug e intercettazione del codice specifico per questo compito, capace di focalizzarsi sulle sole parti indispensabili per l'intercettazione del codice, in grado di ridurre il calo prestazionale attuale.

Le prestazioni pure dell'interprete JavaScript risultano però essere fuorvianti e poco attinenti alla percezione reale che l'utente ha della propria velocità di navigazione. Molto più importante risulta essere infatti il tempo necessario ad una pagina per essere completamente renderizzata all'interno del browser e diventare così usufruibile dall'utente. Sono stati perciò eseguiti dei test volti a misurare il peso relativo dell'estensione rispetto al tempo di caricamento delle pagine. In Figura 5.1 possiamo vedere, riportati graficamente, i risultati di questo test, eseguito su un set di 6 siti di riferimento.

Se ne deduce chiaramente che l'effettivo carico legato all'utilizzo dell'e-

stensione varia moltissimo dal tipo di pagina caricata e da eventuali soluzioni tecnologiche offerte dalla stessa per evitare il caricamento ex-novo di codice già noto/eseguito: più la pagina risulta essere complessa e ricca di codice JavaScript da eseguire, più il caricamento sarà affetto dal ritardo di analisi. In particolare per alcuni siti si è notato il mancato beneficio derivante dal sistema di cache interno dell'estensione browser: questo indica la presenza nella pagina di soluzioni per impedire esplicitamente il manifestarsi di fenomeni di caching aggressivi e pertanto l'estensione non può far altro che continuare ad analizzare il codice ad ogni caricamento (con l'onere connesso) invece di basarsi sui risultati pregressi, causando un crollo prestazionale notevole.

Capitolo 6

Conclusioni e sviluppi futuri

L'obiettivo iniziale della realizzazione di un sistema di identificazione e protezione dalle minacce JavaScript mediante l'analisi statica del codice è stato raggiunto con successo.
E' stato implementato e testato un sistema che consente complessivamente di:

- Raccogliere su vasta scala il codice JavaScript contenuto, incluso e/o generato dinamicamente all'interno delle pagine Web con uno strumento appositamente realizzato e mantenendo il livello di granularità dei sample raccolti al singolo frammento di codice.

- Analizzare il codice in modo completamente statico, al fine di estrarre indicazioni su azioni potenzialmente malevole compiute dal codice stesso.

- Identificare il grado di pericolosità del singolo frammento di codice analizzando le caratteristiche statiche dello stesso, con un grado di affidabilità complessivamente discreto.

- Proteggere attivamente la navigazione dal codice JavaScript, utilizzando il sistema d'analisi e identificazione costruito, con buone ca-

ratteristiche globali e facilmente implementabile ed estendibile in un contesto reale

6.1 Sviluppi futuri

Il sistema fin qui descritto non rappresenta assolutamente un punto d'arrivo nel difficile compito dell'analisi, dell'identificazione e della protezione dal codice JavaScript, quanto una base valida sulla quale procedere.

Per il sistema di analisi ed identificazione, si ipotizzano sviluppi per:

- Revisione caratteristiche analizzate sul codice
  Revisionare, sulla base di eventuali nuovi studi pubblicati in quest'ambito, il sistema di analisi, migliorando la capacità d'estrazione delle caratteristiche del codice, aggiungendo gli eventuali nuovi elementi identificati come potenziali veicoli di minaccia.

- Metodologia di identificazione dei sample simili
  Unire al sistema di hashing diretto sul codice un sistema di firme più generale, capace di identificare frammenti di codice differenti ma caratterizzati dalla medesima struttura architetturale, sia al fine di evitare l'analisi per codice strutturalmente simile, sia per potenzialmente migliorare il sistema di analisi.

- Dataset di riferimento per l'addestramento del classificatore
  Costruire un nuovo dataset per l'addestramento del classificatore, cercando di comporre il più efficacemente possibile un sottoinsieme di sample malevoli validi ed eliminando l'errore di etichettatura che ha afflitto questo sistema.

Per il sistema di protezione, invece, si ipotizzano sviluppi per:

- Realizzazione servizio web di analisi
  Imitando WEPAWET, è possibile costruire un servizio di analisi via

web, per offrire il servizio in forma base a chiunque abbia necessità di analizzare codice separatamente dall'utilizzo reale nel browser.

- Sviluppare l'estensione per il browser Firefox
  Come evidenziato nel Capitolo 5, l'estensione attuale soffre di un calo prestazionale molto marcato, dovuto principalmente all'utilizzo del debugger JavaScript integrato nativamente nel browser. Uno sviluppo futuro potrebbe quindi concentrarsi sull'implementazione ex-novo di un sistema di debug specifico per il compito dell'estensione, al fine di ridurre il peso complessivo dell'estensione sull'esperienza di navigazione.

- Estendere il sistema di protezione ad altri browser
  Sviluppare le necessarie estensioni per l'utilizzo del sistema anche per altri browser, sia per aumentare la copertura del sistema di protezione ma anche per consentire una raccolta dati approfondita utilizzando browser con identità differenti.

Bibliografia

[1] Dan Boneh Collin Jackson Gustav Rydstedt, Elie Bursztein. Busting frame busting: a study of clickjacking vulnerabilities on popular sites, 2012.

[2] Giovanni Vigna Marco Cova, Christopher Kruegel. Detection and analysis of drive-by-download attacks and malicious javascript code, 2006.

[3] Benjamin Livshits Paruj Ratanaworabhan and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks, 2009.

[4] Benjamin Zorn Charles Curtsinger, Benjamin Livshits and Christian Seifert. Zozzle: Low-overhead mostly static javascript malware detection, 2011.

[5] Benjamin Zorn Clemens Kolbitsch, Benjamin Livshits and Christian Seifert. Rozzle: De-cloaking internet malware, 2011.

[6] Thorsten Holz Mario Heiderich, Tilman Frosch. Iceshield: Detection and mitigation of malicious websites with a frozen dom, 2012.