NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

GRADUATE PROGRAM

MASTER THESIS

Detecting drive-by-download attacks on the web

Stefania Varvara I. Martziou

Supervisor: Alexis Delis, Professor NKUA

ATHENS

JULY 2012

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανίχνευση μη-εξουσιοδοτημένης αυτόματης εγκατάστασης κακόβουλου λογισμικού μέσω Διαδικτύου

Στεφανία Βαρβάρα Ι. Μάρτζιου

Επιβλέπων: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2012

MASTER THESIS

Detecting drive-by-download attacks on the web

Stefania Varvara I. Martziou
S.N.: M1092

SUPERVISOR: Alexis Delis, Professor NKUA

EXAMINATION COMMITTEE:

Alexis Delis, Professor NKUA
Aggelos Kiayias Professor NKUA

July 2012

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ανίχνευση μη-εξουσιοδοτημένης αυτόματης εγκατάστασης κακόβουλου λογισμικού μέσω Διαδικτύου

Στεφανία Βαρβάρα Ι. Μάρτζιου
Α.Μ.: Μ1092

ΕΠΙΒΛΕΠΩΝ: Αλέξης Δελής, Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Αλέξης Δελής, Καθηγητής ΕΚΠΑ
Άγγελος Κιαγιάς Καθηγητής ΕΚΠΑ

Ιούλιος 2012

ABSTRACT

In this thesis we define the problem of the unauthorised installation of malware to a users' system during her visit to a web page that has been infected with malware. This problem is very important because the percentage of web pages that can harm the user is increasing while the users' traditional defences, like for example a firewall, are proving unable to cope with this kind of attacks. A number of approaches have been suggested to detect those attacks but even the most technically advanced ones have scaling issues, which is prohibitive when designing systems for the internet. This has created the need for a filter that will be able to quickly reject bening web pages and forward to the more advanced systems those that may contain malevolent code. We present the implementation of such a filter, the rational behind its design and its evaluation.

SUBJECT AREA: System design

KEYWORDS: drive-by-download attack, malware, classification, filter, web, security.

ΠΕΡΙΛΗΨΗ

Στην εργασία αυτή ορίζουμε το πρόβλημα της μη εξουσιοδοτημένης εγκατάστασης κακόβουλου λογισμικού στο σύστημα ενός χρήστη κατά την επίσκεψή του σε μια ιστοσελίδα η οποία έχει μολυνθεί με κακόβουλο κώδικα. Το πρόβλημα αυτό είναι αρκετά σημαντικό καθώς το ποσοστό των ιστοσελίδων που μπορούν να βλάψουν έναν χρήστη ολοένα και αυξάνεται και επίσης οι παραδοσιακές άμυνες του χρήστη, όπως π.χ. το τείχος προστασίας, δεν μπορούν να σταματήσουν τέτοιου είδους επιθέσεις. Πολλές προσεγγίσεις έχουν προταθεί για την ανίχνευση αυτών των επιθέσεων αλλά ακόμα και οι πιο τεχνολογικά εξελιγμένες από αυτές παρουσιάζουν προβλήματα κλιμάκωσης, πράγμα το οποίο είναι απαγορευτικό όταν μιλάμε για συστήματα που αφορούν στο Διαδίκτυο. Αυτό δημιούργησε την ανάγκη για ένα φίλτρο το οποίο θα μπορεί γρήγορα να απορρίπτει ιστοσελίδες οι οποίες δεν βάζουν σε κίνδυνο τον χρήστη και να προωθεί στα πιο εξελιγμένα συστήματα μόνο τις σελίδες αυτές που είναι πιθανόν να περιέχουν κακόβουλο κώδικα. Παρουσιάζουμε λοιπόν την υλοποίηση ενός τέτοιου φίλτρου, τα κίνητρα πίσω από τις επιλογές σχεδιασμού του και την αξιολόγησή του.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Ανάπτυξη συστημάτων και εφαρμογών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: κακόβουλο λογισμικό, φίλτρο, ταξινόμηση, παγκόσμιος ιστός, ασφάλεια.

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να ευχαριστήσω τον καθηγητή κ. Αλέξη Δελή, ο οποίος μου έδειξε πόσο ποιοτική σχέση μπορούν να έχουν οι καθηγητές με τους φοιτητές τους. Χωρίς την πολύτιμη συμβολή και ενθάρρυνσή του δε θα ήταν δυνατή η διεκπαιρέωση της διπλωματικής αυτής εργασίας.

Θα ήθελα επίσης να ευχαριστήσω τον διδάκτορα Θοδωρή Γιαννακόπουλο για τις συμβουλές του και τη βοήθειά του στον τομέα της ταξινόμησης και της εκτίμησης των αποτελεσμάτων.

Contents

List of Figures

List of Tables

1. Introduction

In the early years of the world wide web, when the user base was small and the technology still in its infancy, there where virtually no commercial services or any other way to profit from an online presence. However, in the last years internet usage has proliferated among the computer users. This has led to the advancement of web technologies and the creation of a rapidly growing online economy (such as e-commerce, paid services, etc). Nowadays, it's almost necessary for a company to maintain a web site, offer their services online but also use the internet for their every day operation (email services e.t.c.).

All this progress has attracted a number of people that are trying to illegally profit from this online economy. The main way the attackers are trying to profit is by creating a botnet, i.e. a network of thousands of compromised computers that belong to unsuspecting users and is under control of the attacker. Using a botnet a malicious user can earn thousands of dollars per day. The most common way to create a botnet is to deliver web malware (executable code that is allowing the attacker to control the computer the code runs on) to the users.

There are two main ways that web malware is delivered to the user. The first way of delivering malware is through traditional social engineering techniques, in which the user is tricked to download and execute the malware on his own. This may happen by enticing the user to download free content (like software or pirated content) which contains the malware. The malware is installed and executed by the user.

The second and more devious way to deliver malware is by targeting one or more browser vulnerabilities. When the user visits an infected web page, the malware is automatically downloaded, installed and executed without the user's consent or any other action. This category of techniques is called drive-by-download attacks and is the focus of this thesis [1].

1.1 Drive-by-download attack

As a drive-by-download attack we define every download that is performed without the knowledge and consent of the user (usually a malware or a computer virus). The main reason why these attacks occur is to allow the attacker to infect and take over a network of computers in order to create a botnet, which is usually later used to launch a DDoS attack.

There are two main reasons why we chose to deal with this kind of attacks. The first reason is that traditional methods of protection (firewalls, NATs or proxies) do no work for this kind of attacks because the attacker does not directly target the user. Instead, the infection is masked as a legitimate user request for content, which happens to be malware. None of these methods of protection can distinguish between a "normal" user request and an action by a vulnerable, compromised browser. The second reason became apparent from a study by Google researchers: at least 1.3% of all searches contain one or more malevolent web pages in their results (figure 1.1). Furthermore, out of the top 1 million URLs that appear as search results, more than 6 thousand are malicious web pages. It is apparent that driver-by-download attacks are an important issue and deserves the attention.

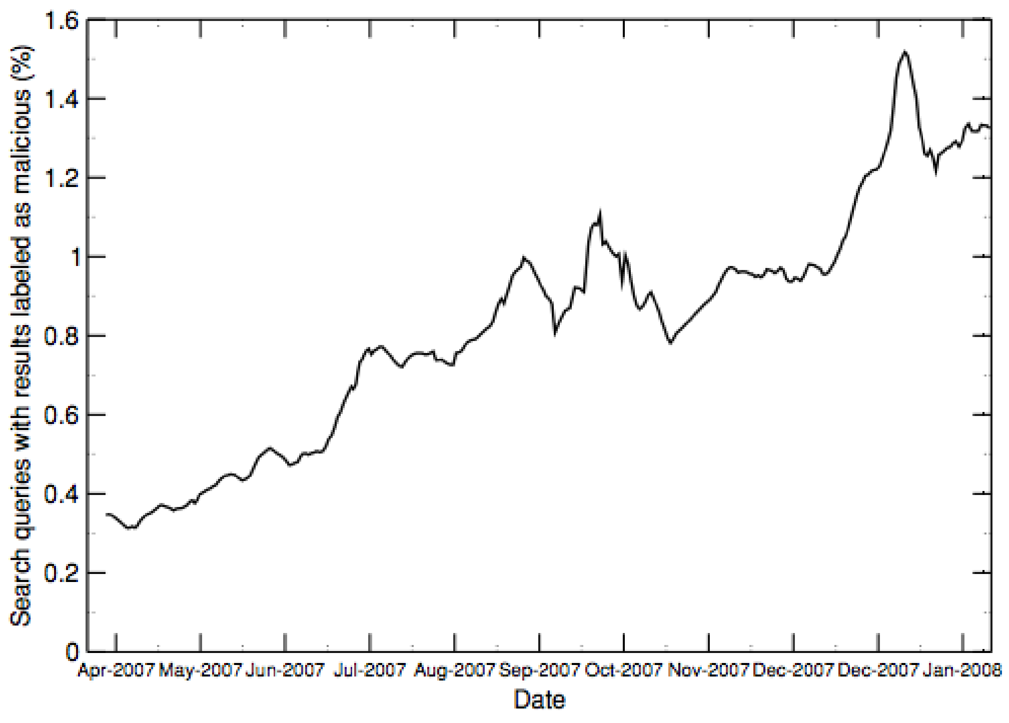


Figure 1.1: 1.3% of all searches contain one or more malevolent web pages in their results [1]

A successful drive-by-download attack requires two prerequisites to have been met: the user's browser must contain at least one vulnerability and the user must visit a malicious web page. Unfortunately, both of the requirements are easy to meet: modern browsers contain millions of lines of code and it is almost certain that a security vulnerability will exist while the number of malicious web pages is very large and surprisingly easy to visit, even accidentally. The main phases of a drive-by-download attack are the following:

- A user visits an infected web site.
- The browser download the initial exploit script.
- The exploit targets a browser vulnerability.
- The exploit code is automatically executed.
- Drive-by-download attack begins.
- The downloaded executable is installed and ran.
- The system is now infected.

In figure 1.2 we can see the steps that are performed during a drive-by-download attack. In this figure as "landing site" we refer to the server containing the infected web page and as "malware distribution site" we refer to the server that is delivering the malware. Also, we note that between the initial visit of the user to the infected web page and the actual download

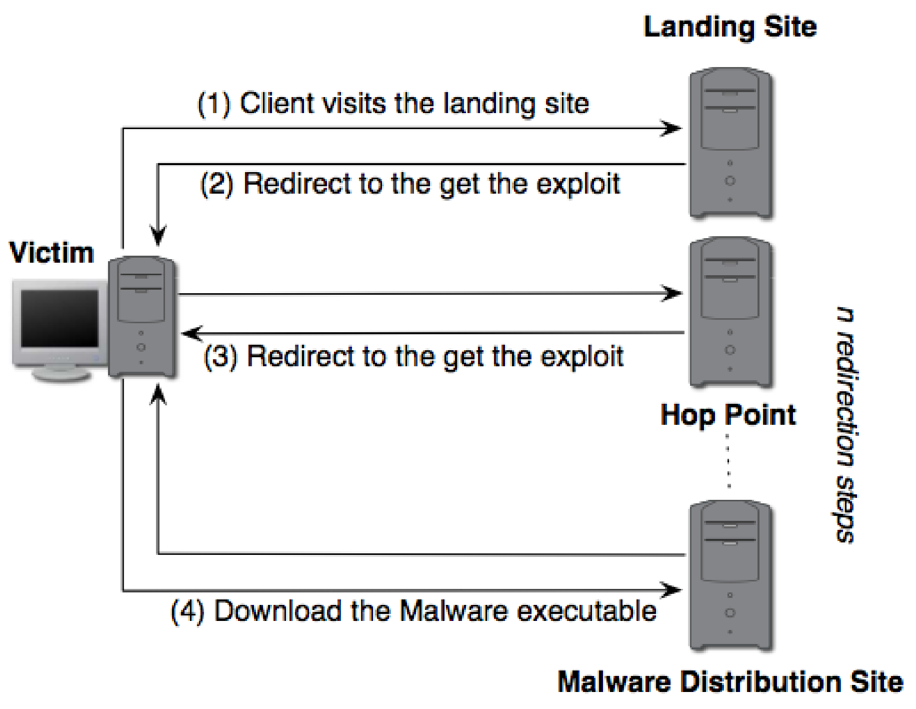

Landing Site
(1) Client visits the landing site
(2) Redirect to the get the exploit
Victim
(3) Redirect to the get the exploit
Hop Point
n redirection steps
(4) Download the Malware executable
Malware Distribution Site

Figure 1.2: Drive-by-download steps [1]

of the malware there may be an arbitrary number of redirects from server to server. This makes the detection and shut down of the malware distribution servers more difficult and time consuming, giving more time to the attacker to spread the malware [1].

1.2 Infection mechanisms

In the previous section we explained how a drive-by-download attack starts, i.e. by a user visiting an infected web site. The existence of legitimate but - somehow - infected web pages is crucial to this kind of attacks because normally users do not tend to visit obscure and unknown pages set up by an attacker. A web site with a lot of traffic (e.g www.cnn.com) that has been compromised will launch many more attacks than a web site that has been created by an attacker just for this purpose, no matter how much effort he puts into it. For this reason, the attackers focus on infecting existing, legitimate web sites and this happens with four different ways [3].

Compromised web server A web site may be written and configured to be as secure as possible and without any vulnerabilities. However, none of these precautions used by the web site creators matter if the web server that is hosting the site is not secure. An attacker may manage to take advantage of a security vulnerability in the server's software (this may happen for any application that runs in the server, like a database server, the http server or even the operating system itself) and thus gain the required privileges to insert the exploit code in the hosted web sites. From that point onwards the web sites will start delivering the exploit code.

Advertising In most cases, advertisements in a web site are not hosted by the same web site. On the contrary, the web site administrators are renting advertisement space in their web pages and (usually using a JavaScript snippet) the advertisements are downloaded directly from the advertisers' servers. Usually, this does not pose a threat since the advertisers rely on the web masters for their work and they are very careful not to deliver malicious code. However, there are many cases where the customer of the advertisement space is not the supplier of the advertisement code. Instead, this space is sold to third party advertisers which have no contact with the web masters. A notable example of this practise is Ads[1], a service by Google that is buying advertisement space from the web masters and sells it to the advertisers.

An attacker may take advantage of such services by buying advertisement space and injecting exploit code to unsuspecting web pages. This leads to the conclusion that the trust between the web masters and the advertisers should not be transitive, i.e. the web masters should not trust blindly the content they are displaying in their web pages even when dealing with a reputable buyer of their advertisement space.

Third party widgets A third party widget is a small piece of code (usually JavaScript) that is hosted in a web site and provides services to other web sites that choose to include it in their pages. Widgets usually enhance the web pages by providing small pieces of information, like the weather, a traffic counter, etc. If the widget is not hosted in a reputable web server, it may deliver malicious code to the visitors of the web site that is including it.

User submitted content The last prevalent mechanism that can be used to infect a legitimate web page is the content that is submitted by end users. This content may come from comments to articles, from posts to user forums, blog entries, etc, and usually is stored in a database or even as a piece of HTML code. If the web pages do not sufficiently control and sanitise the user input, an attacker may be able to inject the exploit code in the web page.

1.3 Related work

Over the years a large number of systems have been created that attempt to locate and identify infected legitimate web pages or pages that have been specifically designed to launch drive-by-download attacks. The methodologies applied by those systems can be broadly classified to four main categories.

Traditional antivirus tools The first approach is based on existing antivirus tools [4]. These tools are using static signatures and try to identify patterns usually present in malicious code.

[1]http://www.google.com/ads/

The main advantage of this method is its speed but it becomes obsolete through the use of advanced obfuscation of the malicious code. Another disadvantage of this approach is that the list of patterns must be regularly updated to include the signatures of newly discovered malicious code and as a result such systems are slow to respond to new threats.

Low interaction honeypots Honeypots are systems that pose as a client and interact with the server to examine whether an attack has occurred. Low interaction honeypots are emulating a web browser and try to locate patterns of malicious responses from the web server (for example, a call to an ActiveX component which contains a buffer overflow vulnerability with a very long argument is a suspicious server behaviour). The problem with this kind of honeypots is that they are preconfigured with the list of suspicious patterns and as a result are unable to identify malicious code for which there are no patterns.

Two notable examples of low interaction honeypots are:

PhoneyC [5] mimics legitimate web browsers and can understand dynamic content by de-obfuscating malicious content for detection. Furthermore, PhoneyC emulates specific vulnerabilities to pinpoint the attack vector. PhoneyC is a modular framework that enables the study of malicious HTTP pages and understands modern vulnerabilities and attacker techniques.

Monkey-Spider [6] a crawler based client honeypot initially utilising anti-virus solutions to detect malware. It is claimed to be fast and expandable with other detection mechanisms.

High interaction honeypots High interaction honeypots are fully functional browsers that run in a controlled environment (usually a virtual machine). Instead of trying to detect a suspicious pattern of code from the web server, they execute the code and try to identify the changes to the state of the system that indicate the execution of malicious code. Such changes may be the creation or alteration of files outside of allowed folders, the modification of the system's registry, the creation of new processes, etc.

The main advantage of these systems is that they are very effective in identifying the results of an attack but they have their share of (serious) disadvantages. First, their performance is quite low since they need to thoroughly inspect the system. Another disadvantage is that they require a successful attack in order to identify a malicious web page. However, many attacks rely on a specific combination of the software of the client (browser, components and even operating system) and as a result, unless the honeypot contains a large number of combinations of client software, the malicious web page may be missed. A final disadvantage is that the attacker may be able to avoid detection by the honeypot either by identifying that the execution environment is a virtual machine and stopping the attack or by delaying the execution of the attack for a period of time (this way the honeypot will detect no attacks right after loading and executing the code of the infected web page but a real system would suffer from the attack a little while later).

A few notable examples of high interaction honeypots are the following:

HoneyMonkey [7] is state based and detects attacks on clients by monitoring files, registry, and processes. A unique characteristic of HoneyMonkey is its layered approach to interacting with servers in order to identify zero-day exploits. HoneyMonkey initially crawls the web with a vulnerable configuration. Once an attack has been identified, the server is reexamined with a fully patched configuration. If the attack is still detected, one can conclude that the attack utilizes an exploit for which no patch has been publicly released yet and therefore is quite dangerous.

Capture-HPC [8] differs from existing client honeypots in various ways. First, it is designed to be fast. State changes are being detected using an event-based model allowing to react to state changes as they occur. Second, Capture is designed to be scalable. A central Capture server is able to control numerous clients across a network. Third, Capture is supposed to be a framework that allows to utilize different clients. The initial version of Capture supports Internet Explorer, but the current version supports all major browsers (Internet Explorer, Firefox, Opera, Safari) as well as other HTTP aware client applications, such as office applications and media players.

HoneyClient [9] was the first open source client honeypot and is a mix of Perl, C++, and Ruby. HoneyClient is state-based and detects attacks on Windows clients by monitoring files, process events, and registry entries. It has integrated the Capture-HPC real-time integrity checker to perform this detection. HoneyClient also contains a crawler, so it can be seeded with a list of initial URLs from which to start and can then continue to traverse web sites in search of client-side malware.

JSAND (JavaScript Anomaly-based aNalysis and Detection) [10] is a novell system with a very good success rate of discovering malicious web pages. JSAND is emulating a web browser and instead of monitoring the changes in the state of the system, it is recording the events that occur during the interpretation of the HTML elements and the execution of the JavaScript code. For each event, it extracts a number of features and evaluates the web page using anomaly detection techniques. This allows the system to identify malicious content even in the presence of previously unseen attacks. JSAND is available in Wepawet[2], an online service where users can submit URLs and files that are automatically analysed, delivering detailed reports about the type of observed attacks and the targeted vulnerabilities.

[2]http://wepawet.cs.ucsb.edu

2. Objectives - Methodology

The objective of this thesis is to create a fast and lightweight filter that is able to distinguish with a very good accuracy a malicious from a benign web page. The web pages that are assumed to be malicious will be analysed by the more powerful and accurate but slow tools that were described in the previous chapter. For this reason it is very crucial for this filter to produce very few false negatives (web pages that are malicious but are labeled as benign), since those will be considered safe for the user and will not be further analysed. However, the production of a false positive (a benign web page that is labeled as malicious) may be undesirable but not as harmful as a false negative: the web page will be further analysed with the slow tools before being declared safe for the user.

The methodology for the creation of the filter is quite simple in principle: define and extract a set of characteristics for the web pages and then based on these characteristics decide whether the web page is malicious or not. However, while simple in principle, this method requires the identification of the correct set of characteristics (or features) and also the use of the correct labeling process.

The set of features that were selected are based on three main components of a web page: the structure of the HTML code of the web page, the structure and content of the JavaScript code of the web page and the URLs (both the URL of the web page and those contained in it). The selected features are described in chapters 3, 4, and 5. The labeling process used comes from the text mining world and is called classification. In short, we use a set of web pages, whose label ("malevolent" or "benign") we know a-priori, to train a classifier and then use the classifier to label unknown web pages. We thoroughly discuss it in chapter 6.

The ideas and methodology of this thesis is based on the work described in [11]

3. HTML features

HTML (short for HyperText Markup Language) is the markup language that is used in the vast majority of web pages. The term "markup" denotes that using this language the author of the web pages is "marking" or annotating the content of the page with information on how it will be presented. This marking is accomplished by the use of "tags" or "elements", which are enclosed in "<" and ">". An example of an element is <b>bold</b>, which means that the word "bold" should be displayed using a bold font. A very basic example of an HTML page follows.

```
<html>
  <head>
    <title>Hello world!</title>
  </head>
  <body>
    <h1>Hello world</h1>
  </body>
</html>
```

There is a very large number of elements in use by HTML but only a few of them are important for our analysis:

html This is the root element of an HTML page.

head Inside this element the author is adding all the metadata of this web page.

body This element contains the actual content of the web page, along with all the elements that describe its format and presentation by the browser.

script A script element contains (or point to) an executable client side script. Most of the times, this script is written in JavaScript.

iframe An iframe is a container in an HTML page whose content is coming form another web page.

object An object element is an extension point in HTML which allows for the inclusion of arbitrary objects in the page.

The features that are extracted from the HTML code are based on the statistics of the raw content of the page but also on information that is based on the structure of the HTML code.

3.1 Side effects of malicious code injection

An attacker that has managed to compromise a web page does not always have full control on the resulting code. For example, the result of HTML injection most of the times affect a small portion of the HTML code. This usually leads to malformed HTML structure with repeating elements (e.g, repeating <head> or <body> elements) or elements not in the correct place (e.g, <iframe> tags in the <head> section of the HTML code). As a result, the number of such mistakes is one of the HTML features used. Another feature that might indicate that the web page is compromised is the total number of characters in the page.

3.2 Excessive presence of code

Another feature of many compromised web pages is the presence of excessive amount of JavaScript code compared to the actual content. The percentage of the JavaScript code is one the measured features. Another feature is the number of <script> elements, which also indicates the presence of malicious code.

3.3 Suspicious code

The last set of HTML features involves the detection of suspicious code. We count all the potential attempts by an attacker to hide the inclusion of malicious code in a web page. These features are:

Number of <iframe> elements Iframes can be used to download and execute malicious code.

Number of hidden elements Most of the elements in a compromised web page containing malicious code are not visible to the end user (i.e. by using the "hidden" attribute) as an attempt by the attacker to hide the changes.

Number of elements with small size . Many of the drive-by-download exploits do not use visibility attributes (e.g, display="none") to hide the malicious elements but instead use direct size attributes (width, height). Because of that, we count the number of elements that could contain malicious code (<div>, <iframe> or <object> elements) and have a total area below a certain threshold.

Number of <object> and <embed> elements Both of those two elements are used to include and execute an external application and many times are used by attackers to inject their malicious code. The number of such elements in a web page is another feature we measure.

Presence of suspicious objects We examine whether the class id of the included <object> elements belongs to lists of ActiveX components with known vulnerabilities.

Presence of meta refresh tags A common pattern in drive-by-download attacks is the inclusion of a meta refresh tag in a compromised web page. A meta refresh tag is an HTML element which instructs the browser to reload the web page, potentially redirecting to an entirely different web page. The existence or not of this tag is one of the features.

Number of included URLs or Code in external domain Elements whose content is specified by using a URL instead of containing it directly (e.g, <script>, <iframe>, <embed> etc) could point to compromised servers. Because of this, the number of included URLs is one of the selected features and we also count the subset of these URLs that point to an external server.

Number of malicious patterns Another feature we extract is the number of patterns known to exist in compromised web page. One example of such patterns is a meta tag redirecting to an exploit server.

Number of elements with suspicious code Using a simple heuristic, we are able to decide if the content of an element is considered malicious code or not. The number of elements whose code is considered to be shell code is yet another feature.

Percentage of whitespace Shell code usually contains a much smaller percentage of whitespace compared to normal HTML content or benign JavaScript code. For this reason, the percentage of whitespace in HTML is another feature we extract.

Presence of scripts with wrong extension A common method used by attackers to hide their attempt to download and execute malicious code is to use the wrong filename extension in their JavaScript code, hoping that the antivirus programs will ignore this file. For this reason, the presence of such an attempt is alarming and is considered a feature.

Percentage of unknown tags The existence of unknown tags in the HTML code may indicate that the attackers target a browser's inability to safely ignore those elements and their content and thus having an exploitable vulnerability. The percentage of unknown versus standard tags is another feature we extract.

4. JavaScript features

Javascript is a dynamic programming language commonly used by web browsers that enables a richer user experience. It allows for much more dynamic content than plain static HTML pages, since it gives the web programmers the ability to interact with the user, perform asynchronous communication with the server, control the browser and modify the structure of the HTML document.

Since JavaScript is a powerful and versatile language and JavaScript code may be downloaded from any web site, modern browsers have taken precautions to prohibit as much as possible the execution of malicious code or, at least, prevent the undesired consequences in case malicious code is indeed executed. The two main measures browsers take are sandboxing and the same origin policy. Sandboxing is the restriction imposed by the browser to the operations that a script may perform: modifying the contents of the web page or performing asynchonous calls to the server are permitted since they are normal operations but accessing the hard disk or data from other web pages is not allowed. The code of each web page is given the least possible rights that it needs in order to perform "normal" and expected actions. The same origin policy is the restriction that the browser imposes to scripts, that allows to only access the DOM of the pages that originate from the same server but not that of pages from other servers. Usually, when a security vulnerability is discovered this happens due to violations of one or both of the above measures [12].

JavaScript code is included in a web page using one of two methods. The first one is by explicitly using a <script> element, which either contains directly the code or points to a location in the server. The second way to load code is taking advantage of the dynamic nature of the language that allows a script to load an arbitrary string, interpret it as code and execute it at will.

When extracting features from the JavaScript code of a web page we follow two main kinds of code analysis. The first is the statistical analysis of the code. Using statistical analysis we examine the contents of the code, e.g. the length of strings, the presence of certain strings, the number of whitespace characters, etc. The other kind of analysis is the analysis of the AST (Abstract Syntax Tree) of the code. The AST is a tree representation of the syntactic structure of the program, with each node representing one construct of the code. An AST contains limited information about the code under inspection but this information is enough for our purposes and has the additional advantage that an AST is relatively fast to build.

4.1 Injection of malicious code

The first category of the features extracted from the JavaScript code of a web page involves the identification of the attempts of an attacker to inject the malicious code in the web page. This usually happens by modifying the structure of the web page and adding new elements that in turn download and execute the malicious code. For this reason, we extract a number of features:

Number of DOM manipulating functions The number of JavaScript functions that attempt to insert a new element in the DOM tree of the web page.

Occurrences of the string "iframe" Another feature is the number of occurrences of the "iframe" string in the JavaScript code. The reason we extract this feature is that the word "iframe" in the JavaScript code indicates that the script may by attempting to modify the DOM tree of the web page in order to inject malicious code.

Number of suspicious tag strings Sometimes, instead of trying to inject malicious code by using iframes, the attackers may choose to use other elements, like "script", "object", "embed" or "frame". For this reason we also count the number of the occurrences of these tags.

Number of suspicious object names Just like in section 3.3, we count the number of suspicious object names that occur in the JavaScript code.

Number of suspicious strings We count the number of strings like "evil", "shell", "spray" etc, that are commonly found in malicious, non obfuscated JavaScript code.

4.2 Attempts to execute code

After forcing the user's browser to download the malicious code, the next step is to cause its execution. This is detected by the presence of the following features:

setTimeout(), setInterval() functions . These functions are used to delay the execution of a function or to schedule a piece of code to be repeatedly executed. Since they can be used by an attacker to execute code that was injected using the previous techniques, we count the number of occurrences of each function.

Event attachments Drive-by-download attacks usually wait for the whole web page to be loaded before starting the code execution. Moreover, attackers try hard to mask the presence of the malicious code, so they even try to disable the error reporting when the attack is not successful. For these reasons we count the number of event attachments but only for the events that are generated when a web page is loaded (onload) or when an error occurs (onerror).

eval() function . The eval() function is used to interpret an ordinary string as JavaScript code and then execute it. Since this is the primary means of a drive-by-download attack to download and then execute the malicious code, we count the number of occurrences of the eval() function [13] [2].

4.3 Suspicious code

Browser detection Another set of JavaScript functions that are usually used by the attackers are functions that allow them to detect the kind of browser that the user is using. This allows the attacker to select which browser vulnerability to target and which malicious code to download and execute. For this reason, we count the number of occurrences of such functions (e.g, navigator.userAgent()).

Keywords-to-words ratio Usually, malicious JavaScript code contains very few keywords compared to the number of strings used. Most of the commands involve mostly variable instantiation, arithmetic operations and function calls.

4.4 Obfuscation

Another major feature of the JavaScript code included in compromised web pages is the attempt of the attacker to obfuscate the code. Obfuscation is the technique that aims to transform the human readable source code to a form that is very difficult for a human to read and understand. In our case this is very important since JavaScript is an interpreted language and as such, the included JavaScript code is not downloaded in a binary format; instead, the source code is downloaded and then executed as is by the browser. Therefore, it is crucial for the attacker to hide the malevolent code. However, the presence of obfuscated code is not by itself an evidence that the code is malicious. A large number of benign web sites also use obfuscating techniques on their JavaScript code in order to protect it from copyright violations [3].

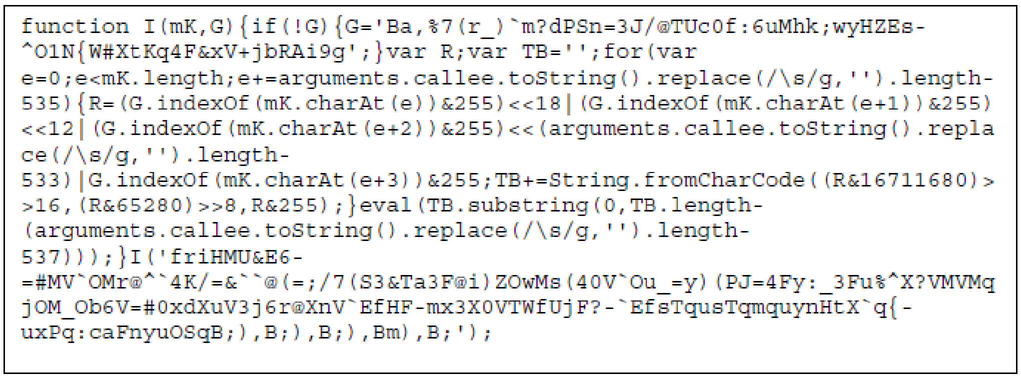
In order to detect the existence of obfuscated code, we extract the following features:

Entropy of strings and of the script as a whole Obfuscated strings with no apparent meaning or expressed in other forms (like ASCII codes or unicode values) tend to contain repeating characters. Because of this, the entropy of the strings, the script as a whole, and the maximum entropy of all the script's strings are three of the extracted features [14].

Length of strings One of the main features of obfuscated JavaScript code is the fact the the length of the strings tends to be much larger than that of the non obfuscated code. A string is considered to be a "long string" if its length exceeds a certain threshold. This threshold is decided during the training phase. The features we extract in this case are the number of long strings in the JavaScript code and the maximum and average length of the script's strings. Another feature we extract is the number of long variable variables or function names [14].

Average script line length Obfuscated JavaScript code tends to contain really long lines of code, especially compared to "normal", benign code. For this reason, we calculate the average line length.

Probability of a script to contain shellcode Another feature we extract is the probability of a script to contain shellcode (shellcode is the code that is executed using a browser



```
function I(mK,G){if(!G){G='Ba,%7(r_)`m?dPSn=3J/@TUc0f:6uMhk;wyHZEs-
^O1N{W#XtKq4F&xV+jbRAi9g';}var R;var TB='';for(var
e=0;e<mK.length;e+=arguments.callee.toString().replace(/\s/g,'').length-
535){R=(G.indexOf(mK.charAt(e))&255)<<18|(G.indexOf(mK.charAt(e+1))&255)
<<12|(G.indexOf(mK.charAt(e+2))&255)<<(arguments.callee.toString().repla
ce(/\s/g,'').length-
533)|G.indexOf(mK.charAt(e+3))&255;TB+=String.fromCharCode((R&16711680)>
>16,(R&65280)>>8,R&255);}eval(TB.substring(0,TB.length-
(arguments.callee.toString().replace(/\s/g,'').length-
537)));}I('friHMU&E6-
=#MV`OMr@^`4K/=&``@(=;/7(S3&Ta3F@i)ZOwMs(40V`Ou_=y)(PJ=4Fy:_3Fu%^X?VMVMq
jOM_Ob6V=#0xdXuV3j6r@XnV`EfHF-mx3X0VTWfUjF?-`EfsTqusTqmquynHtX`q{-
uxPq:caFnyuOSqB;),B;),B;),Bm),B;');
```

Figure 4.1: A fragment of obfuscated JavaScript code [2]

vulnerability). The probability of a string containing shellcode is calculated by examining the percentage of non-printable ASCII characters, the percentage of hex characters, the length of script in characters and also the percentage of repeating patterns of characters, which also suggest that the string is obfuscated. Another feature is the percentage of whitespace of the code.

4.5 Deobfuscation

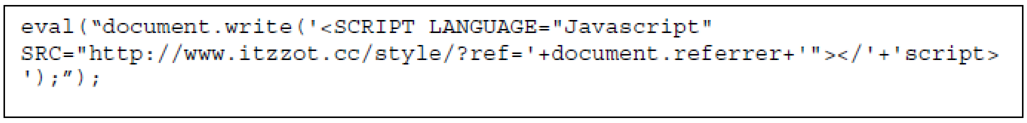
The reverse process of obfuscation is deobfuscation. When the obfuscated JavaScript code has been successfully downloaded in the user's browser, it has to be reverted to "normal" code in order to be executed. A standard way to deobfuscate the code consists of transforming the strings from their obfuscated format to plain text and repeatedly manipulate them with string manipulation functions in order to end up with human readable and thus executable code. We extract the following features to detect the presence of deobfuscation attempts:

Built-in functions commonly used for deobfuscation We count the number of JavaScript built-in functions that are usually used for transformation to plain text: unescape(), fromCharCode(), etc are some of these functions [7] [2].

Presence of decoding routines We examine whether the JavaScript code contains fragments of code that resemble decoding routines. More precisely, we examine the AST of the code to detect loops inside which a long string is manipulated. The presence of such routines is another feature we extract.

String modification functions The last feature of this category is the number of string modification function used in the script. These functions (replace(), substr(), substring(), etc) are usually used in the manipulation of the long strings of the previous case [7].

Direct string assignments . We count the occurrences of all possible ways that can be used in JavaScript to assign a value to a string. Usually, the deobfuscation and decryption procedures link to a large number of string assignement commands [2].



```
eval("document.write('<SCRIPT LANGUAGE="Javascript"
SRC="http://www.itzzot.cc/style/?ref='+document.referrer+'"></'+'script>
');");
```

Figure 4.2: Deobfuscation of figure reffig:js-obs reveals an attempt to indirectly load JavaScript code [2]

5. URL features

The last broad category of features consists of the characteristics of the URL of the web page and also the URLs that are included by the web page. The reason why we don't examine only the URL of the web page (something that happens when trying to detect phising attempts) is that a compromised web page may have a "safe" URL but the code that has been injected may point to malevolent URLs.

A URL, short for Uniform Resource Locator, is a reference to a resource, along with enough information on how to retrieve this resource. The syntax of a URL (even though some parts may be omitted) is

<scheme>://<domain>:<port>/<path>?<query>#<fragment>.

The parts of a URL are the following:

scheme Describes the protocol used to retrieve the resource. The most common ones used in the web are http and https, but many others are also available, like ftp, file, mailto, etc.

domain Describes the hostname of the IP address of the server that is hosting the resource. A URL may lack a host name and instead use the actual IP address of the hosting server.

port The port number that is used to connect to the server. It is an optional attribute and, if omitted, the default value for the protocol is used (e.g. 80 for http, 443 for https, etc).

path The path describes the location of the resource within the hosting server. It may refer to an actual path in the filesystem or to an arbitrary path, interpreted programmatically by the server.

query The query string is optional and contains additional information that is passed to the software running in the server.

fragment The fragment (also an optional attribute) describes a position within the resource, such as a specific part of a web page.

We perform four types of analysis on the URLs of a web page:

5.1 Syntactical

The first type of analysis deals with the format of the URL and relies purely on the syntactical features of the URL under inspection. Some of the extracted features are the domain name length, whether the original URL is relative, presence of suspicious domain name, length of the filename, and presence of port number.

Host based obfuscation The URL of the vast majority of bening web pages do not contain host obfuscation, while a large number of malicious web pages are host obsfuscated with an IP address [15].

Absolute/relative length of URL Analysis has shown that legitimate URLs contain a '/' right after or very close to the organisation name. For example, `http://www.di.uoa.gr/announcements/undergraduate` contains a '/' right after uoa.gr. In average, bening URLs have 0.21 characters between the organisation name and the path separator (with a maximum of 14 characters) while phising URLs contain an average of 7 characters between the organisation name and the path separator (with a maximum of 63 characters) [15].

Top level domain The reason for examining the top level domain of the URL is the alarming contribution of Chinese-based web sites to the web malware problem: overall, 67% of the malware distribution servers and 64% of the web sites that link to them are located in China [1].

Absence of subdomain Many malicious web pages are referring to the content distribution servers without indicating a subdomain (i.e., the web page refers to "abc.com" instead of "www.abc.com").

Presence of IP address Moreover, many malicious web pages are not related to a domain name but are referenced through their IP address. This may happen because the web page is hosted in a machine of a compromised public network.

Presence of suspicious patterns Many of the malicious URLs has been found to contain a number of patterns that may indicate that the attackers used some existing exploitation kits. A list of patterns was compiled based on these kits and is compared against the URLs under inspection. The presence of any of these patterns in the URL is a feature.

5.2 DNS based

Another set of features is extracted by analysing the DNS entries of the URL under inspection. Domain Name Service (DNS) is an internet service whose main responsibility is to translate domain names to IP addresses. It is a hierarchical service, with each node of the service being responsible for a subset of the IP address space. The DNS protocol is a classic client-server protocol. The clients are mainly the user's web browsers that are trying to locate the servers that contain the web pages and also mail transfer agents that are trying to forward emails to their recipients.

Each DNS server maintains a list of resource records each of which contains information associated with domain zones and IP addresses. There is a large number of resource record types, the most notable of which (and of interest to us) are the following:

A (address) record which contains an IP address and is usually used to map a hostname to the IP address of the respective host.

NS (name server) record which is used to delegate the DSN query to the authoritative name server.

MX (mail exchange) record which maps a domain name to a list of message transfer agents for that domain. This record is used when forwarding emails to their recipients.

PTR (pointer record) which maps an IP address to a hostname. This record is normally used for reverse lookups, i.e. when the user is searching for the hostname that corresponds to a given IP address.

For each URL we are querying for the A, NS and MX records and for each kind of records we extract the following features.

First IP address Analysis shows that the malware distribution sites are concentrated in a limited number of / prefixes. About 70% of the malware distribution sites have IP addresses within 58.* -- 61.* and 209.* -- 221.* network ranges [1].

Number of IP addresses Usually, the A, NS and MX records for bening sites contain more than one IP addresses. This happens for fault tolerance and load balancing. The records for malicious web sites usually contain only one IP address.

TTL of IP addresses Usually, a short time to live of an IP entry indicates that the entry is meant to be short lived, either because the malicious web page will be moved to another host or the malicious content distribution server will be shut down.

ASN of IP addresses Research by Google indicates that the IPs of the malware distribution sites belong to a relatively small subset of ASNs (around 500 in 2011) [1].

Furthermore, besides the examination of each record individually, we also extract two more features:

Resolved PTR record Bening web pages usually have correct and complete DNS entries. For a bening page, the PTR lookup for the host name will return the IP address of the page. On the contrary, malicious web pages usually do not have PTR records in the DNS entries.

Consistency between PTR and A records Another, more strict, criterion is the request for a resolved PTR record to be equal to the IP address of the A record of the web page. This usually holds for benign pages and not for malicious ones.

5.3 WHOIS based

Whois is a widely used Internet record listing that identifies who owns a domain and how to get in contact with them. The Internet Corporation for Assigned Names and Numbers (ICANN) regulates domain name registration and ownership. Whois records have proven to be extremely useful and have developed into an essential resource for maintaining the integrity of the domain name registration and website ownership process.

A Whois record contains all of the contact information associated with the person, group, or company that registers a particular domain name. Typically, each Whois record will contain information such as the name and contact information of the Registrant (who owns the domain), the name and contact information of the Registrar (the organisation or commercial entity that registered the domain name), the registration dates, the name servers, the most recent update, and the expiration date. Whois records may also provide the administrative and technical contact information (which is often, but not always, the registrant).

From all this information we care about the registration, last update and expiration dates of the web page's domain under inspection. Usually, malicious sites have relatively recent registration date and/or their expiration date is in the near future. This happens because attackers often purchase domain names for small periods of time, expecting to be discovered quickly [16].

5.4 GeoIP based

GeoIP is a service that provides geographic information about a user based on her IP address, such as the city, state, country, longitude and latitude. This service is provided by a number of companies, usually for a fee even though usually free versions are also available.

Using the GeoIP databases, we extract the Country code, region, timezone and network speed of the web page under investigation. Besides the reasons stated above (the vast majority of malicious sites residing in China), a study by the University of Indiana shows that many phising domains are not hosted on the country they were registered in [16].

6. Decision making

Each individual feature described in the previous chapters only provides a hint that something may be suspicious about a web page. However, unless they are all combined together, they cannot provide enough evidence that the web page is indeed malicious. The problem is that with such a large number of features, each one with it's own diverse type of metrics, it is very difficult to evaluate the whole set of features of a web page.

The solution to this problem comes from data mining and, more specifically, classification. Classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations whose category membership (class) is already known. The individual observations are analysed into a set of quantifiable properties, known as features.

An algorithm that implements classification, especially in a concrete implementation, is known as a classifier. The term "classifier" sometimes also refers to the mathematical function, implemented by a classification algorithm, that maps input data to a category. In the terminology of machine learning, classification is considered an instance of supervised learning, i.e. learning where a training set of correctly identified observations is available.

In our case, we defined two classes of web pages: the class of benign and the class of malevolent web pages. A classifier is configured using a set of pre-classified web pages, the class of which is known a-priori. The classifier is then used to assign each new, unknown web page to a class.

One other choice that had to be made was what kind of classifiers to use. Since we wanted to test more than one classifiers to make sure that the performance of the filter wasn't affected by a bad choice of classifier, we chose to train and evaluate 2 classifiers: the Random forest and J48 classifiers. A brief description of the two classifiers follows.

J48 classifier is an implementation of the C4.5 algorithm for the Weka framework and it works by building a decision tree from the training data. C4.5 is the successor to the ID3 algorithm proposed by Ross Quinlan.

A decision tree is a predictive machine-learning model that decides the target value (dependent variable) of a new sample based on various attribute values of the available data. The internal nodes of a decision tree denote the different attributes, the branches between the nodes tell us the possible values that these attributes can have in the observed samples, while the terminal nodes tell us the final value (classification) of the dependent variable.

In each node in the tree the algorithm is choosing the attribute that splits the samples in subtrees with the smallest possible entropy (or the maximum information gain). In other words, it tries to split the samples in subsets containing the fewest possible classes. It then continues recursively in each subset until all the samples in the subset belong to the same class, the list of features has been depleted or the subset is empty.

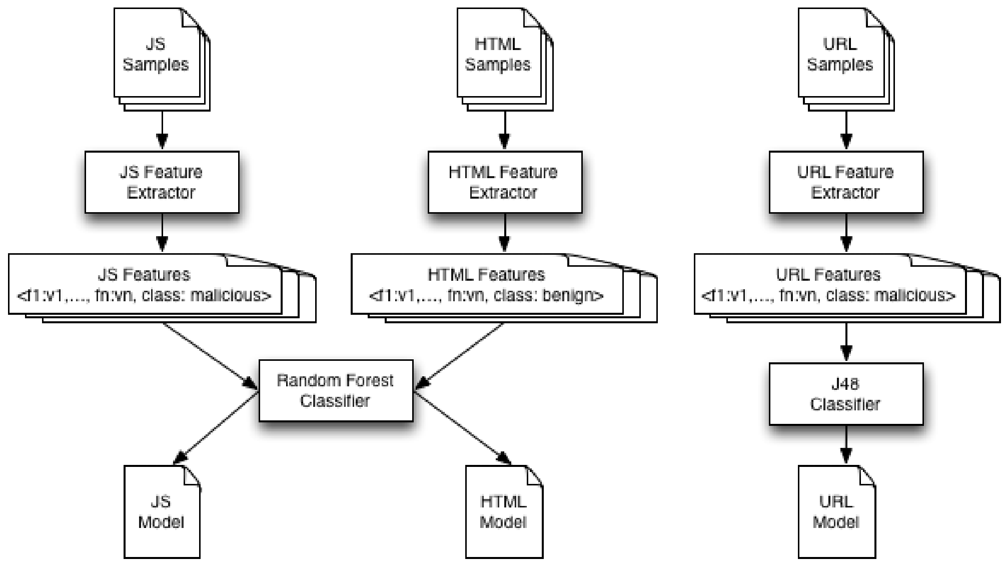


Figure 6.1: Classifier training using the classifiers with the best performance

Random Forests are an ensemble learning method (also thought of as a form of nearest neighbour predictor) for classification and regression that construct a number of decision trees at training time and outputting the class that is the mode of the classes output by individual trees (Random Forests is a trademark of Leo Breiman and Adele Cutler for an ensemble of decision trees).

Random Forests are a combination of tree predictors where each tree depends on the values of a random vector sampled independently with the same distribution for all trees in the forest. The basic principle is that a group of "weak learners" can come together to form a "strong learner". Random Forests are a wonderful tool for making predictions considering they do not overfit because of the law of large numbers. Introducing the right kind of randomness makes them accurate classifiers and regressors.

6.1 Training

In order to train a classifier we need a relatively large set of web pages, the class of which is already known, that belong to both classes. We extract all the features from each web page and the results are used to create a vector. The vectors from all known are given as training input to the classifier, along with the class of each vector. The output of this process is a model that is used in the classification of unknown web pages (see figure 6.1).

6.2 Classification

After the classifier is trained using the set of known web pages, it is ready to be used to identify unknown web pages either as benign or malevolent. The process of classification is similar to the one of training: The features of the unknown web page are extracted and placed in a vector and along with the model that resulted from the training process are given

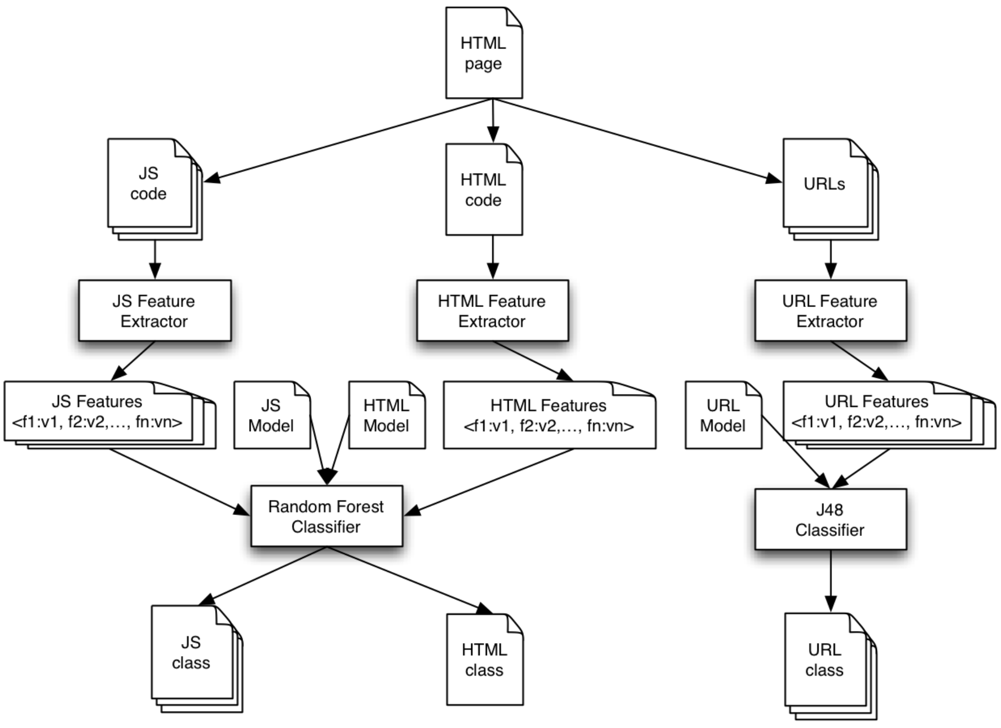


Figure 6.2: Classification using the classifiers with the best performance

as input to the classifier. The output of the classification is the class that the web page is assigned to. Hopefully, with the correct set of the training set and the correct classifier, the class of the web page is the correct one (see figure 6.2).

7. Implementation

7.1 Evaluation

We used a training set of 400 malicious and 1800 benign JavaScript scripts and evaluated two different classifiers: the Random Forests and J48 classifiers (both of which have been described in the previous chapter).

When trying to evaluate the performance of a classification system and compare the two or more classifiers we use a number of metrics. The most frequently used and visually useful is the confusion matrix, but others exist like the recall precision and overall accuracy. A brief description of these metrics follows:

Confusion matrix contains information about actual and predicted classifications done by a classification system. Performance of such systems is commonly evaluated using the data in the matrix. The following table shows the confusion matrix for a two class classifier.

The entries in the confusion matrix in 7.1 have the following meaning in the context of our study:

- a is the number of correct predictions that an instance is benign
- b is the number of incorrect predictions that an instance is malicious
- c is the number of incorrect of predictions that an instance is benign
- d is the number of correct predictions that an instance is malicious

	Benign	Malicious
Benign	a	b
Malicious	c	d

Table 7.1: A sample confusion matrix

The rest of the metrics are the following:

Overall accuracy (AC) is the proportion of the total number of predictions that were correct. It is determined using the equation:

$AC = \frac{a+d}{a+b+c+d}$

Recall (R) is the proportion of positive cases that were correctly identified, as calculated using the equation:

$R = \frac{d}{c+d}$

Precision (P) is the proportion of the predicted positive cases that were correct, as calculated using the equation:

$P = \frac{d}{b+d}$

Cross-Validation is a statistical method of evaluating and comparing learning algorithms by dividing data into two segments: one used to learn or train a model and the other used to validate the model. In typical cross-validation, the training and validation sets must cross-over in successive rounds such that each data point has a chance of being validated against. The basic form of cross-validation is k-fold cross-validation. In k-fold cross-validation the data is first partitioned into k equally (or nearly equally) sized segments or folds. Subsequently k iterations of training and validation are performed, such that within each iteration a different fold of the data is held-out for validation while the remaining k-1 folds are used for learning. The most usual value of k and also the one that we used is 10.

Random Forests classifier

	Benign	Malicious
Benign	0.8130	0.0018
Malicious	0.0153	0.1697

Table 7.2: Random Forests confusion matrix

From the confusion matrix in 7.2 we can easily calculate and the rest of the metrics for the Random Forests classifier: the overall accuracy is 98.28%, the recall is 99.77% for benign and 91.69% malicious scripts and the precision is 98.14% for benign and 98.92% for malicious scripts.

J48 classifier

	Benign	Malicious
Benign	0.8037	0.0115
Malicious	0.0180	0.1667

Table 7.3: J48 confusion matrix

From the confusion matrix in 7.3 we can easily calculate and the rest of the metrics for the J48 classifier: the overall accuracy is 97.04%, the recall is 98.59% for benign and 90.22% malicious scripts and the precision is 97.80% for benign and 93.54% for malicious scripts.

Conclusion

Comparing the results of the two classifiers, it is obvious that for the JavaScript samples the Random Forests classifiers outperforms the J48 classifier. The false positives (benign scripts that were classified as malicious) are only 0.02% but the percentage of false negatives (malicious scripts that were not correctly identified) is high, at 8.3%. However, this percentage will be dramatically reduced when we train and use classifiers for the other two categories of features (HTML and URL).

7.2 Implementation

The entire code of this thesis was written in Java and a large number or external libraries was used to implement all the functionality. The most notable of these libraries are:

HTMLParser [17] is a library that parses HTML and tries to produce the DOM tree. However, it doesn't perform any kind of error correction and reports all errors in the structure of the HTML code. This library was used to detect anomalies in the structure of the page and extract the corresponding HTML features.

CyberNeko HTML parser [18] is another library that parses HTML but it tries to overcome mistakes in the structure of the HTML code and create a DOM tree. It was used to extract the rest of the HTML features.

Rhino JavaScript [19] engine is a standalone JavaScript execution engine that does not require the presence of a web browser. It was used to parse the JavaScript code and extract the corresponding JavaScript features.

dnsjava [20] is a library that was used in order to extract the DNS features.

MaxMind API and database where used to extract the GeoIP and ASN related features [21].

WEKA API [22] was used to perform the training, classification and evaluation of the classification models.

7.3 Future work

The first step in continuing the work of this thesis is to acquire and use samples of benign and malicious HTML pages and URLs. This will allow us to train and evaluate the classifiers for these sets of features and draw a final conclusion on their performance.

One important optimisation that would be interesting to investigate is the attempt to minimise the number of web pages and JavaScript scripts that we analyse. While the product of this theses is designed to act (or plans to) as a fast filter for a more thorough and much more time consuming analysis tool, its performance should be optimised. One of the obvious ways to achieve this is to avoid repeating the analysis on web pages that have already been analysed. This can be efficiently accomplished by maintaining a signature of the DOM of HTML pages or the AST of JavaScript scripts. The challenge is to identify a suitable signature method and similarity function. The obvious hashing algorithms (like md5) may seem appealing because of their simplicity and speed but have one disadvantage: a single modification (however small) in the web page or script structure result in completely different signatures. A suitable signature method is a tool that will allow us to optimise the performance of this filter.

Finally, the ultimate goal is to create a complete back-end analysis system that will be able to completely filter out the false positives and perform dynamic analysis on the remaining web pages to correctly identify the malicious ones.

Bibliography

[1] Provos N, Mavrommatis P, Rajab MA, Monrose F. All Your iFRAMEs Point to Us. In: Proceedings of the 17th Conference on Security Symposium. SS'08. Berkeley, CA, USA: USENIX Association; 2008. p. 1--15. Available from: http://dl.acm.org/citation.cfm?id=1496711.1496712.

[2] Feinstein B, Peck D. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. In: DEFCON 15; 2007. .

[3] Provos N, McNamee D, Mavrommatis P, Wang K, Modadugu N. The Ghost in the Browser Analysis of Web-based Malware. In: Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets. HotBots'07. Berkeley, CA, USA: USENIX Association; 2007. p. 4--4. Available from: http://dl.acm.org/citation.cfm?id=1323128.1323132.

[4] ClamAV Team. Clam Antivirus; Jul 2012 (accessed February 3, 2012). Available from: http://www.clamav.net/.

[5] Nazario J. PhoneyC: A Virtual Client Honeypot. In: Proceedings of the 2Nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More. LEET'09. Berkeley, CA, USA: USENIX Association; 2009. p. 6--6. Available from: http://dl.acm.org/citation.cfm?id=1855676.1855682.

[6] Ikinci A, Holz T, Freiling F. Monkey-Spider: Detecting Malicious Websites with Low-Interaction Honeyclients. In: In Proceedings of Sicherheit, Schutz und Zuverlassigkeit; 2008. .

[7] Wang Y, Beck D, Jiang X, Roussev R. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In: NDSS; 2006. .

[8] Seifert C, Steenson R. Capture - Honeypot Client (Capture-HPC). Victoria University of Wellington, NZ; 2006 (accessed April 21, 2012). Available from: https://projects.honeynet.org/capture-hpc.

[9] HoneyClient Project Team. HoneyClient; Jul 2012 (accessed April 21, 2012). Available from: http://www.honeyclient.org/.

[10] Cova M, Kruegel C, Vigna G. Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code. In: Proceedings of the 19th International Conference on World Wide Web. WWW '10. New York, NY, USA: ACM; 2010. p. 281--290. Available from: http://doi.acm.org/10.1145/1772690.1772720.

[11] Canali D, Cova M, Vigna G, Kruegel C. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In: Proceedings of the 20th International Conference

on World Wide Web. WWW '11. New York, NY, USA: ACM; 2011. p. 197--206. Available from: http://doi.acm.org/10.1145/1963405.1963436.

[12] Moshchuk A, Bragin T, Gribble SD, Levy HM. A Crawler-based Study of Spyware on the Web. In: NDSS; 2006. .

[13] Choi Y, Kim T, Choi S, Lee C. Automatic Detection for JavaScript Obfuscation Attacks in Web Pages Through String Pattern Analysis. In: Proceedings of the 1st International Conference on Future Generation Information Technology. FGIT '09. Berlin, Heidelberg: Springer-Verlag; 2009. p. 160--172. Available from: http://dx.doi.org/10.1007/978-3-642-10509-8_19.

[14] Byung-Ik K, Chae-Tae I, Hyun-Chul J. Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation. International Journal of Advanced Science and Technology. 2011;26:19--31. Available from: http://www.sersc.org/journals/IJAST/vol26/2.pdf.

[15] Garera S, Provos N, Chew M, Rubin AD. A Framework for Detection and Measurement of Phishing Attacks. In: Proceedings of the 2007 ACM Workshop on Recurring Malcode. WORM '07. New York, NY, USA: ACM; 2007. p. 1--8. Available from: http://doi.acm.org/10.1145/1314389.1314391.

[16] Ma J, Saul LK, Savage S, Voelker GM. Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '09. New York, NY, USA: ACM; 2009. p. 1245--1254. Available from: http://doi.acm.org/10.1145/1557019.1557153.

[17] Oswald D. HTMLParser; Jul 2012 (accessed April 18, 2012). Available from: http://htmlparser.sourceforge.net/.

[18] A Clark MG. CyberNeko HTML Parser;. http://nekohtml.sourceforge.net/.

[19] Mozilla Foundation. Rhino Project; Jul 2012 (accessed April 23, 2012). Available from: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.

[20] Wellington B. dnsjava; Jul 2012 (accessed April 22, 2012). Available from: http://www.dnsjava.org.

[21] MaxMind Inc . MaxMind GeoIP Databases; Jul 2012 (accessed April 24, 2012). Available from: https://www.maxmind.com/en/geoip2-databases/.

[22] Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The WEKA Data Mining Software: An Update. SIGKDD Explor Newsl. 2009 Nov;11(1):10--18. Available from: http://doi.acm.org/10.1145/1656274.1656278.