POLITECNICO DI MILANO
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Automated Collection and Analysis of Runtime-Generated Strings in a Web Browser

Relatore: Prof. Stefano Zanero
Correlatore: Ing. Federico Maggi

Tesi di Laurea di:
Manuel Fossemò
Matricola n. 734531

Anno Accademico 2010-2011

To the victims of the 2011 Tohoku earthquake

Acknowledgments

During the preparation of this thesis, a terrible earthquake and its consequences killed many people in Japan. My preferite hobbies are playing videogames and reading manga comics, so I must thank the japanese people for all their incredible artworks. The dedication of this thesis goes to them. Never give up!

Now I would like to thank all the people near me that gave me love, friendship and support.

First of all, thank you very much Chiara for all your patience. I hope that our future together will be always beautiful with many "scucciolis".
Thank you dad and mom for giving anything I ever wanted.
Thank you Marco even if you took me many times at breaking point.
Thank you Ennio, Piera and Paolo for threating me like a son and a brother in the last 10 years.
Thank you Stefano, Andrea, Pierpaolo, Giampaolo, Alessandro and Mario because even if we live faraway our friendship never changed.
Thank you Yanick, Davide, Marco, Massimiliano because you never gave me a punch in the face for my bother during the university lectures.
Thank you Matteo, Lorenzo and Mattia. I enjoyed very much living together like a family.
Thank you all UIC italian and international friends. I will never forget those 4 months in Chicago!!
Thank you Stefano and Federico for your incredible help and patience during the preparation of this work.

Contents

1 Introduction **1**

2 State of the Art **4**
2.1 JavaScript Malware 4
2.1.1 Buffer overflows 4
2.1.2 Drive-by Download and Heap Spraying Attack 6
2.2 Related Work 9
2.2.1 JSAND 9
2.2.2 Prophiler 10
2.2.3 Zozzle 10
2.2.4 Mitigating Heap-Spraying Attacks 11

3 Features of JavaScript String Variables **13**
3.1 Features 13
3.1.1 Feature 1: Distribution of Strings Length 14
3.1.2 Feature 2: Presence of Obfuscation or Code Generating Functions 14
3.1.3 Feature 3: Count of Reserved Words 15
3.1.4 Feature 4: Ratio of the number of collected variables to the number of different referrers 15
3.2 Implementation Details 16
3.2.1 Large Scale Collection 16
3.2.2 Modified JavaScript Interpreter 16
3.2.3 JavaScript Strings Collection 18
3.2.4 Storage 21

3.2.5 Creation of baseline dataset with Google Safe Browsing 21

4 Experimental Results **24**

4.1 Feature 1: Distribution of Strings Length 24

4.2 Feature 2: Presence of Obfuscation or Code Generating Functions 35

4.3 Feature 3: Count of Reserved Words 37

4.4 Feature 4: Ratio of the number of collected variables to the number of distinct referrers 41

4.5 Other Features 42

5 Conclusions **46**

Bibliography **48**

List of Figures

2.1 Stack canaries 5

3.1 Example of collected strings 19

4.1 CDF length good local global 25
4.2 CDF length good property concatenated 26
4.3 CDF length malicious local global 27
4.4 CDF length malicious property concatenated 28
4.5 CDF length good local global 29
4.6 CDF length good property concatenated 30
4.7 Comparation local global 32
4.8 Comparation property concatenated 33
4.9 Analysis of functions in substring 36
4.10 Analysis of JS reserved words in substring 38
4.11 Analysis of JS reserved words in substring 39
4.12 Analysis of JS reserved words in substring 40
4.13 Ratio of the number of collected variables to the number of different referrers 41
4.14 Comparison of composition 43
4.15 Analysis of characters in substring 45

Listings

2.1 Code snippet used for heap spraying. 7
2.2 Example of obfuscated code. 8
3.1 C code inserted into `jsinterp.cpp` used to collect JS strings allocated on client-side by the browser 18
3.2 Code for collecting string 20
3.3 Code for saving strings on MySQL database 22
3.4 Code for filter the malicious set with GSB 23

List of Tables

4.1 Parameters of the good strings collected 44
4.2 Parameters of the malicious strings collected 44
4.3 Parameters of the GSB strings collected 44

Sommario

Analizzando le recenti minacce informatiche perpetrate via web si nota un maggiore impiego di tecniche di attacchi che sfruttano le variabili (in particolare le stringhe) JavaScript come vettore del codice macchina malevolo. Quando il browser della vittima interpreta il codice JavaScript della pagina malevola, il codice macchina viene eseguito ed ha luogo l'attacco. In letteratura sono stati proposti diversi approcci per rilevare questo tipo di attacchi, ma nessuno di questi si è rivelato abbastanza veloce da poter essere integrato all'interno di un browser commerciale, a causa del tempo necessario a valutare i parametri usati per riconoscere il comportamento malevolo. Aspetto cruciale dei metodi più efficaci sinora proposti è la corretta scelta delle feature per discriminare codice malevolo da codice non malevolo. Per questo motivo, in questo lavoro ci siamo focalizzati sulla raccolta ed analisi delle variabili allocate durante l'esecuzione del browser sulla macchina dell'utente che visita la pagina web. Per raccogliere queste variabili abbiamo instrumentato Firefox, il browser di Mozilla, e con queste variabili valutiamo alcuni parametri cercando di scoprire quelli che possono aiutarci a distinguere il comportamento di una pagina normale da quello di una pagina malevola, a prescindere dal tipo di attacco che la pagina malevola lancia quando un browser la visita.

Abstract

In the last years web based threats that exploit memory vulnerabilities use JavaScript variables (in particular strings) as vectors of malicious machine code. When the victim's browser interprets JavaScript code of the malicious page, machine code is executed and the attack happens. In literature some approaches have been proposed to detect heap spraying attacks, but none of them is enough lightweight to be integrated inside a commercial browser due to the time necessary to evaluate the features used to recognize malicious behavior. A very important aspect of the more effective methods proposed is the correct choice of the features to use to distinguish malicious code from non malicious code. For this, in this thesis work we focus on collecting and analyzing the variables that are generated at runtime by the browser when it visits a web page. To collect the variables we instrumented Mozilla Firefox and with this data we evaluated some features to understand if it is possible to distinguish malicious web pages from good web pages, aside from the kind of attacks they launch.

Chapter 1

Introduction

Modern web pages are quite sophisticated and very often incorporate a large amount of client-side code (JavaScript being the most popular language) that provide rich functionalities (e.g., opening new windows, validating input values of a web form, or changing images as the mouse cursor moves over them). Unfortunately, JavaScript is abused by attackers to spread malicious code and gain control of the victim machines. Client-side attacks usually exploit vulnerabilities of the browser or of one of its plugins, and attempt to download and execute shellcode, or steal sensitive information. Today's attacks have migrated from memory-based vulnerabilities (e.g., stack buffer overflows) to heap spraying attacks, because while for memory-based attacks exist compiler techniques to mitigate their effects (e.g., StackGuard [4]), for heap spraying attacks many exist solutions, that will be explained later in this thesis work, but none of them is lightweight enough to be integrated into commercial browsers [5]. There are two approaches to determine the maliciousness of a piece of code: dynamic analysis and static analysis. Dynamic analysis is performed at runtime, during the execution of the scripts found in web pages, whereas static analysis is performed without running the code of the scripts. The main advantage of static analysis over dynamic analysis is that, given enough memory and time, all possible execution paths can be taken into account. However obfuscation and encryption techniques are easy ways to prevent efficient static analysis [6]. On the other hand, with dynamic

analysis, the problem is, as we said before, that it is not possible to determine if all execution paths have been examinated. Despite this, dynamic analysis permits the examination of obfuscated code, a common technique in modern web pages. For this reason, this work focuses on techniques of dynamic analysis.

More precisely, this work aims at determining whether it is possible to cast a decision on the maliciousness of a web page by analyzing the variables (from hereinafter referred to as "JavaScript strings" or simply "JS strings") that are allocated dynamically by the JavaScript interpreter of the browser. Specifically, we concentrate on the following features:

- Distribution of string length, to determine if strings containing malicious code are longer than normal JS strings;
- Presence of obfuscation or code generating functions, to determine if some specific functions have an higher presence inside malicious strings;
- Count of reserved words, to determine if a malicious string is recognizable by counting the number of reserved JS words it contains;
- Ratio of the number of collected variables to the number of different referrers.

It is important to collect a big number of strings so that we can conduct robust experiments to (dis)confirm our intuition. To achieve our goal we have modified Spidermonkey [1], the Mozilla Firefox browser's JavaScript engine, to save all the JS strings allocated by a web page at runtime. We choose Firefox because it is open source and so we could put our hands on the source code. In particular, we used the Spidermonkey modified as discussed in [7], and we further change the code in order to collect the JS strings on an output file. We proceed as follows:

1. run Firefox on an big number of URLs and collect a huge quantity of JS strings;
2. save the collected strings on a DB;

3. analize the strings with queries to evaluate useful features.

In summary, we obtained the following results:

- Malicious JS code tends to allocate more strings with length greater than 20 with respect to normal sites;
- Words used inside obfuscating or code generating functions are found in malicious strings with an higher percentage;
- Variables allocated by malicious JS code have a high occurrence of some JS reserved words;
- Good JS code allocates more variable with respect to malicious JS code.

This work is organized as follows:

- In Chapter 2 there is an overview about JS attacks and security. Some related work will be explained, with particular attention on the limits they have and the features they use;
- Chapter 3 shows the implementation details, with code snippets, partial input and output and technical issues;
- In Chapter 4 are shown and explained our experimental results;
- In Chapter 5 we draw the conclusions and we propose some possible improvements of our work.

Chapter 2

State of the Art

In this chapter we present how JS strings can be abused as attack vectors and we analyze some solutions proposed to mitigate the effects of those attacks.

2.1 JavaScript Malware

In this section we present the main vulnerabilities and techniques that can be exploited to produce web attacks.

2.1.1 Buffer overflows

Buffer overflows are classic security vulnerabilities that have been around since the beginning of programming, and are still occurring everywhere today[1]. A buffer overflow is where memory has been overwritten on the stack, that is the memory area where the original value is replaced by the values of other variables such as arrays or strings. Typically this can occur by sending a very large amount of data to the application and then injecting malicious code at the end of this large data. JavaScript strings are a good vector for malicious users to inject code into the application. When this code has been placed on the stack, sometime later this information will be pulled off of the stack and executed. In this way, a malicious user can exploit buffer overflows to execute arbitrary code on the victim's machine.

[1] `http://www.testingsecurity.com/how-to-test/buffer-overflows`

Buffer Overflows Mitigation

A number of solutions have been developed to inhibit malicious stack buffer overflow exploitation. One method consists in detecting that a stack buffer overflow has occurred and thus prevent redirection of the instruction pointer to malicious code.

Such method leverages stack canaries. Stack canaries [4] work by placing a small integer, called canary, the value of which is randomly chosen at program start, in memory just before the stack return pointer. Most buffer overflows overwrite memory from lower to higher memory addresses, so in order to overwrite the return pointer (and thus take control of the process) the canary value must also be overwritten. This value is checked to make sure it has not changed before a routine uses the return pointer on the stack. If the canary value has changed, than the compiler raises a signal. Figure 2.1 shows where the canary is placed in the stack memory.

An approach to prevent malicious user to launch program is to enforce

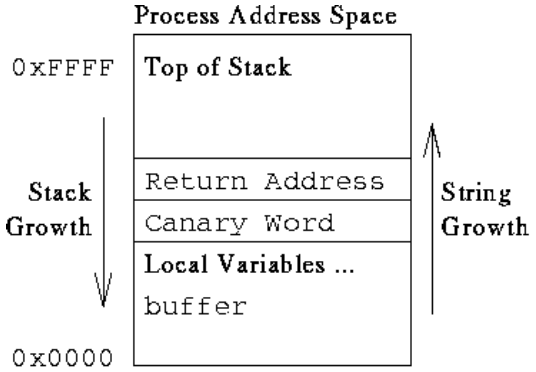

Figure 2.1: The canary word is placed before the return address of the function to prevent the overwrite of the return address value. If the buffer grows and overwrites the canary word, the compiler raises a signal.

memory policy on stack memory region to disallow execution from the stack. This means that in order to execute shellcode from the stack an attacker must either find a way to disable the execution protection from memory, or find a way to put his shellcode payload in a non-protected region of memory. This method is becoming more popular now that hardware support for the no-

execute flag is available in most desktop processors. To avoid this protection, the attackers started to store shellcode in unprotected memory regions like the heap.

2.1.2 Drive-by Download and Heap Spraying Attack

A drive-by download (from hereinafter referred to as "DbD") is any download of software that happens without the knowledge and consent of a user. Such attacks usually follow this scenario. First, the attacker loads a sequence of executable instruction (i.e., shellcode) into the address space of the web browser. This is often done using JavaScript strings, because is very important that the instructions are stored at sequential addresses in memory. The second step exploits vulnerabilities in the browser or a plugin that allows the attacker to divert the control flow of the application to the shellcode. Injecting and exploiting code in the heap is more difficult for an attacker than placing code on the stack because the addressers of heap objects are less predictable than those of stack objects. Therefore, to make their exploits more reliable, attackers resort to a technique called heap spraying.

Heap spraying creates multiple instances of the shellcode, combined with NOP sledge. More precisely, a heap-spraying attack populates the heap with a large number of objects containing the shellcode, assigning the exploit to jump to an arbitrary address in the heap, and relying on luck that the jump will land inside one of the objects containing the shellcode. NOP sledges are used to increase the likelihood that the attack will succeed. An heap spraying code snippet is shown in Listing 2.1, in which we can see how the variables used to fill the heap memory are created:

Drive-by Download Mitigation

In the last few years several new approaches have been proposed to protect users from DbD attacks. It is possible to split these approaches into two groups: dynamic approaches and static approaches.

Dynamic approaches use honeypots (i.e., systems that imitate the activities of the real systems that host a varieties of services) to visit web pages.

```
1  if (version < 8)
2  {
3  var addkk = unescape("%uA164%u0018....");
4  var mem_array = new Array();
5  var cc = 0x0c0c0c0c;
6  var addr = 0x400000;
7  var sc_len = addkk.length * 2;
8  var len = addr - (sc_len+0x38);
9  var yarsp = unescape("%u9090%u9090");
10 yarsp = fix_it(yarsp, len);
11 var count2 = (cc - 0x400000)/addr;
12 for (var count=0; count < count 2; count++)
13 {
14 mem_array[count] = yarsp + addkk;
15 }
16 var overflow = unescape("%u0c0c%u0c0c");
17 while(overflow.length < 44952) overflow += overflow;
18 this.collabStore=Collab.collectEmailInfo({subj:"", msg:
      overflow});
19 }
```

Listing 2.1: Code snippet used for heap spraying.

In high-interaction honeypots the analysis use traditional browsers and detect signs of a successful DbD attack (e.g., change in file system, registry or running processes), whereas low-interaction analysis emulate browsers and detect the manifestation of an attack (e.g., the invocation of a vulnerable method in a plugin). These approaches usually yield good detection rates with low false positive ([3] and [7]), but, on the other hand, this analysis can be slow, because of the time required by the browser to retrieve and execute all the contents of the web page.

Static approaches rely on the analysis of the static aspects of a web page (e.g., textual context, features of its HTML and JS code). String signature are used by traditional antivirus tools to identify malicious pages. Unfortunately signature can be evaded using obfuscation (Listings 2.2).

Several approaches have focused on statically analyzing JS code to identify malicious web pages. The most common features extracted from scripts are the presence of redirects, the presence of functions used for obfusca-

```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b);
     printf(b); }
    else switch(m1-=h){
    case f:
     a=(b=(c=(d=g)<<g)<<g)<<g;
     return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
     for(a=f;a<j;++a)
      if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
      case g:
       if(n<h)return(g);
       if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
       else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
       if((b=n)>=e)
        for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
       return(o[b-g]%n+k-h);
      default:
       if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
       for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char
          *)m1);
    }
}
```

Listing 2.2: Example of obfuscated code.

tion/deobfuscation, calls to *eval()* function and the presence of shellcode-like strings. In this work we focused on the analysis of the variables (strings)

allocated by JS code, and not on JS code itself, because we are interested to characterize malicious behavior of a web page aside from the kind of attacks it launchs. In addition, we have evaluated the features on large scale to make robust the evaluation of the features, trying to mitigate the effects of the outliers. Static analysis is difficult because, in addition to code obfuscation and runtime code generation (that are very common in both benign and malicious code), the attacker (who knows the list of features being used by each approach) can evade [8].

2.2 Related Work

This section summarize the three most recent tools that have been developed to mitigate DbD attacks, with attention on the features used and limitation they have.

2.2.1 JSAND

JSAND [3] (JavaScript Anomaly-based aNalysis and Detection) is a tool to detect DbD attacks by using machine learning and anomaly detection. Anomaly detection is based on the hypothesis that malicious activity manifests itself through anomalous system events. To select the features to use, the authors have determined the steps that are often followed in carrying out an attack: redirection and cloaking, deobfuscation, environment preparation, and exploitation. Some of the features used by *JSAND* are: number and target of redirections, number of dynamic code executions (e.g., `eval` and `setTimeout`), length of dynamically-evaluated code, number of likely shellcode strings and number of instantiated components (e.g., plugins).

To evaluate these features *JSAND* creates some models to assign a probability score to each feature. A model can operate in training mode, to determine the threshold to distinguish between normal and anomalous feature, and in detection mode, where the established models are used to determine an anomaly score for each observed feature value. The tools caused no false positive (i.e., no good page was flagged as malicious) and 0.2% percent of

false negative (i.e., undetected malicious pages).

To avoid detection an attacker could bypass the features used by the tool [8], but in general, the tool is capable of detecting previously-unseen attacks. Another evasion technique is to check differences between JSAND's emulated environment and a real browser's environment, but it is possible to set up JSAND's environment so that it behaves very similar to a real browser.

2.2.2 Prophiler

Prophiler [2] is a fast filter for large-scale detection of malicious web pages. It analyzes static features of HTML pages embedded JS code, and associated URLs using a number of models that are derived using supervised, machine-learning techniques. Pages that are likely malicious are further analyzed with dynamic detection tools (e.g., [3]). Static analysis is fast, as the web page being analyzed is not rendered and no scripts are executed. The tool extracts features from: HTML and JS code, and the page's URL. For the features include: number of elements with small area, number of suspicious objects, number of included URLs, number of long strings, shellcode presence probability, number of suspicious URL patterns and presence of IP address in URL.
Some of these features could be evaded by malicious scripts, for example by generating DOM elements dynamically. This is however a limitation of any static analysis approach.

2.2.3 Zozzle

Zozzle [5] is a mostly-static detector that examines JS code and decides whether it contains heap spray exploits. It is mostly static because its analysis is entirely static, plus a lightweight runtime component to "unroll" obfuscated or dynamically generated JS. *Zozzle* is integrated with the browser's JS engine, which collects and processes JS that is created at runtime. This is useful to mitigate techniques used against static analysis, as explained in Section 2.2.2. To make the prediction on the behavior of web pages, *Zozzle* uses Bayesian classification of hierarchical features of the JS abstract syntax

tree (AST), which is a tree representation of the abstract syntactic structure of JS code. Specifically, a feature consists of two parts: the context in which the feature appears (e.g., a loop, conditional, function) and the text (e.g., unescape, addbehavior) of the AST node. To keep only a limited number of features, for performance reasons Zozzle only extracts features from expressions (e.g., "`var spray = 54884 - shellcode.length * 2`") and variable declaration. In addition, as most of the variable declarations are not informative (e.g., they are correlated with neither benign nor malicious training sets), features used for classification are chosen with a Chi Square statistic.

Automatic features selection typically yields many more features as well as some features that are biased toward benign JS code, unlike hand-picked features, which are all characteristic of malicious JS code. Some examples of hand-picked features are: `try:unescape`, `function:addbehavior`, `loop:spray`; sample of automatically selected features are: `loop:scode`, `function:anonymous`, `loop:shellcode`. The limitation of *Zozzle* is that, as other classifier-based tools, can be evaded by exploiting the inner workings of the tool and the list of features being used. Another problem is that *Zozzle* produces false positives, although these are less then 1%, their cost is considerably higher than cost of false negatives.

2.2.4 Mitigating Heap-Spraying Attacks

Egele et al. in [7] proposed a technique that relies on x86 emulation to identify JS string buffers that contain shellcode. Their tool is integrated into the browser and it performs the detection before control is transferred to the shellcode. The basic idea is to check string variables allocated by the browser while executing JS scripts and look for those that contain shellcode. Shellcode detection is performed with *libemu*[2], a small library that offers basic x86 emulation: the key intuition is that, if the string contains shellcode, *libemu* would be able to execute it. The tool produces no false positives for a known-good dataset, although false positive may occur in four cases: when the DbD attacks make no use of memory exploits; when exploits client-side

[2] `http://libemu.carnivore.it/`

code other than JS;when malicious code is spread into several scripts that reside in different files; when a threat was injected into the known-malicious dataset.

Chapter 3

Features of JavaScript String Variables

The goal of our work is to perform an evaluation of JS string features. The difference with previous work is that (1) we use features that are generic with respect to the attacks, and (2) we perform a large scale evaluation.

As mentioned in Section 2.1.2, attackers use client-side scripting code to load the shellcode into memory and execute the exploit against vulnerable component. In general, the shellcode is stored inside a JS variable (string). We are interested in strings because it is important that the bytes constituting the shellcode are stored at successive addresses in memory. Otherwise, these bytes would not be interpreted as valid x86 instruction. In JS, the only way to guarantee that bytes are stored in adjacent cells is by using a string variable [7]. Of course it is possible to split the shellcode into smaller segments and than to concat these pieces, but at least the segments must be stored into strings, and so they are collected equally to a normal JS string.

3.1 Features

We concentrate on the following features.

3.1.1 Feature 1: Distribution of Strings Length

This feature is evaluated to check if good sites and malicious sites allocate variables with different length. We first analyze the shape of the cumulative distribution function of the length of the strings grouped by their origin (i.e., local, global, property or concatenated) and type (i.e., good or malicious), and after that we compare the results to search for differences. We chose to evaluate this feature because the machine code stored inside the strings must be quite long to have the desired effect. So we expect that strings that contains machine code are longer than normal string or they are splitted into smaller strings. In the second case, we expect that malicious sites allocate a bigger number of concatenated strings with an higher length.

For example, on a variable such as `PROP` (origin) `http://www.avg.com/homepage` (referrer) `http://www.avg.com/script/jquery.js` (script name) `document.write(script=1);` (string content) `2011/03/12 19:01:41` (datetime of capture) the value of this feature is 25.

3.1.2 Feature 2: Presence of Obfuscation or Code Generating Functions

This feature is evaluated to verify the presence inside the allocated strings of functions used to obfuscate or to dynamically generate malicious code (e.g., `eval`, `document.write`, `unescape`, `concat`). For this purpose we choose some specific words and we count the number of occurrences of each word. We used our intuition and past experience to choose the functions that are most used for obfuscation and dynamic generation of code in order to reduce the set of words to check.

For example, on a variable such as `PROP` (origin) `http://www.avg.com/homepage` (referrer) `http://www.avg.com/script/jquery.js` (script name) `document.write(script=1);` (string content) `2011/03/12 19:01:41` (datetime of capture) the value of this feature is `count(document.write)+=1`.

3.1.3 Feature 3: Count of Reserved Words

This feature is evaluated by counting the number of JS reserved words inside the allocated strings. During the collection of the strings we noticed that many of them were composed only by JS code and so we count the occurrences of specific JS word to verify if some words appears more in malicious variables with respect to good variables. To reduce the set of reserved words we chose the ones that have an higher probability to be correlated with a malicious behavior instead of a normal behavior, such as words used for obfuscation, dynamic code generation or buffer overflows.

For example, on a variable such as `PROP` (origin) `http://www.avg.com/homepage` (referrer) `http://www.avg.com/script/jquery.js` (script name) `unescape("%u9090%u9090");` (string content) `2011/03/12 19:01:41` (datetime of capture) the value of this feature is `count(unescape)+=1`.

3.1.4 Feature 4: Ratio of the number of collected variables to the number of different referrers

This feature is evaluated by calculating the ratio of the collected variables to the number of different referrers (i.e., the URL of the page from which have been originated the collected variables). The number of different referrers is very significative because many sites present in our benign and malicious sets are not active, so the analysis of those sites doesn't produce anything. So the number of different referrers can be used instead of the number of active sites, even if this number is not completely equal to the number of active sites because two different variables can have the same referrer, and so we can verify if good JS code allocate more or less string with respect to malicious JS code.

For example, on a variable such as `PROP` (origin) `http://www.avg.com/homepage` (referrer) `http://www.avg.com/script/jquery.js` (script name) `unescape("%u9090%u9090");` (string content) `2011/03/12 19:01:41` (datetime of capture) the value of this feature is `count(distinctreferrers)+=1`, if `http://www.avg.com/it-en/homepage` was not counted before.

3.2 Implementation Details

In this chapter we explain how our tool works, showing the partial input and output of each part and commenting every code snippet.

3.2.1 Large Scale Collection

To evaluate useful features with a good accuracy we chose to collect a large amount of JS strings. We also save the name of the script that generated each strings, the referrer, the set to which the analyzed web page belongs (i.e., good or malicious), the date of capture and the origin of the strings (i.e., local, global, property or concatenation). These other fields are useful because we can study the evolution of a web page by checking the strings allocated in different period of time, or we can control if the similar variables allocated by the JS code of a web page are created by very different scripts.

To decrease the influence of outliers in our analysis, we collected a large number of strings. For this purpose, we have launched our modified browser on several sites. We used Alexa[1] most popular 10,000 sites to collect known-good strings, whereas for malicious sites we used the blacklists from Malware Domain List[2] (about 3131 URLs), Malware Patrol[3] (about 1556 URLs), Malware Domain Blocklist[4] (about 8299 URLs) and Url Blacklist[5] (about 21,888 URLs). We checked more than 35,000 malicious sites and we were able to collect only 10,051,176 strings because sites were often closed right after they were spotted.

3.2.2 Modified JavaScript Interpreter

To collect the strings we chose to modify Spidermonkey [1], Mozilla Firefox's JS engine, but our approach apply to other browsers. We selected Firefox because it is open source and so we could modify its code. We modified

[1] `http://www.alexa.com/`
[2] `http://www.malwaredomainlist.com/`
[3] `http://www.malware.com.br/`
[4] `http://www.malwaredomains.com/`
[5] `http://www.http://urlblacklist.com/`

`jsinterp.cpp`, which is a class of Spidermonkey [1]. We added our code in the interpreter in four points where strings variables are created:

- strings declared as local inside a JS function;
- string declared as global by the JS code;
- strings that are properties of objects;
- strings that are the result of a concatenation of strings.

We distinguish the concatenated strings because they are useful for some features. Listing 3.1 shows the code inserted in each point. Line 6-9 create the variable `date` with the time of capture of the string in the format `Year:Month:Day Hour:Minutes:Seconds`, useful for save the date in the format `SQL DATETIME`. Line 12 saves into the variable `str3` the first four character of the filename of the script that generates the string, because it is important to check if the filename starts with "`http`" (line 14). This is done to filter out all the variables that aren't allocated by the visited web page (e.g., the variables created by browser such as the version of the plugins installed), that are recognizable because they have filename that starts with "`chrome://`", so we compare the variable `str3` with the string "`http`"(line 13) to collect only the strings that are allocated by the web page. Line 14 also filters out the strings with length 0, because they aren't interesting for our work. A proof that we only collect strings originated by the web page, comes from lines 15-20. This code creates the variable `jsstring` that contains the location of the referrer of the string. In fact the strings we collect have referrer of type "`http://...`" whereas for strings that have other source the location is "`undefined`". Lines 23-25 replace from the collected string the characters "\n", "\r" and "\t" with "_" because they create troubles when we save the strings on the DB. Finally line 26 writes on the output file (defined at line 4, in our case "`fireout.txt`") the collected string, starting with `CONCAT` (`LOCAL`, `GLOBAL`, `PROP` respectively), followed by the referrer, the script's filename, the actual string and the date of capture. An example of output is shown in Figure 3.1:

```
1  FILE * myfile;
2  unsigned int  length1;
3  length1 = JS_GetStringLength(str);
4  myfile = fopen ("/home/manuel/fireout.txt","a");
5  if(myfile!=NULL){
6    char date[20];
7    time_t t = time (NULL);
8    strftime (date, sizeof (date), "%Y/%m/%d␣%H:%M:%S",
        localtime (&t));
9    JSString *str3, *str4;
10   str3=JS_NewStringCopyN(cx,cx->fp->script->filename,4);
11   str4=JS_NewStringCopyN(cx,"http",4);
12   if(length1!=0 && JS_CompareStrings(str3,str4)==0){
13     jsval val;
14     JSObject *glob;
15     glob = JS_GetGlobalObject(cx);
16     JS_GetProperty(cx, glob, "location", &val);
17     JSString *jsstring;
18     jsstring = JS_ValueToString(cx, val);
19     char *mystr;
20     mystr=JS_EncodeString(cx,str);
21     for(j=0;j<length1;j++){
22        if(mystr[j]=='\n' || mystr[j]=='\r'||
             mystr[j]=='\t') mystr[j]='_';
23     }
24     fprintf(myfile, "CONCAT\t%s\t%s\t%s\t%s\n",
          JS_EncodeString(cx,jsstring),
25     cx->fp->script->filename, mystr, date);
26   } fclose (myfile);}
```

Listing 3.1: C code inserted into `jsinterp.cpp` used to collect JS strings allocated on client-side by the browser

3.2.3 JavaScript Strings Collection

To run the browser on each URL we used the script in Listing 3.2: we open the input file (line 25) with the list of URLs to analyze, then with the function `fork()` (line 29) we create a child process. This child process opens our modified browser with the function `execl()` (line 31), giving as inputs the directory of the executable of the browser (in our case `/home/mozilla-1.9.1/objdir-ff-release/dist/bin/firefox`) and the variable

```
1 LOCAL http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1.16)
    Gecko/20110311 Shiretoko/3.5.16 2011/03/12 19:01:41
2 PROP http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    1.9.1.16 2011/03/12 19:01:41
3 CONCAT http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    script1299952901426 2011/03/12 19:01:41
4 LOCAL http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    script1299952901426 2011/03/12 19:01:41
5 PROP http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js none
    2011/03/12 19:01:41
6 PROP http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    text/javascript 2011/03/12 19:01:41
7 CONCAT http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    window.script1299952901426 2011/03/12 19:01:41
8 CONCAT http://www.avg.com/it-en/homepage
    http://www.avg.com/stc/tpl/crp/script/jquery.js
    window.script1299952901426=1; 2011/03/12 19:01:41
```

Figure 3.1: For each string is collected the origin, referrer, script name, content and datetime of capture.

`line` that contains the URL to analyze. The parent process waits for 12 seconds and then kills the child process with the command "`killall -15 firefox-bin`". It is possible that the parent process kills the child process when it is writing the string on the output file (e.g., the loading of the web page analyzed starts a few seconds before the `kill` command). In this case the string on the file could be chopped and it is important to remove this string because it will crash the script in Listing 3.3, which stores the strings into a DB. However, in our case this happened less than 10 times on more than 53 millions of strings collected.

```
1  void wait ( int seconds )
2  {
3    clock_t endwait;
4    endwait = clock () + seconds*CLOCKS_PER_SEC;
5    while (clock() < endwait) {}
6  }
7
8  int main(int argc, char**argv)
9  {
10 int i=0;
11 char * line = NULL;
12 size_t len = 0;
13 int seconds=12;
14 char cmd2[] = "killall␣-15␣firefox-bin";
15 ssize_t read;
16 FILE * infile;
17 FILE * outfile;
18 infile = fopen (argv[1],"r");
19 if (infile!=NULL && outfile!=NULL)
20  {
21  while ((read = getline(&line, &len, infile)) != -1){
22  int pid = fork();
23  if (pid==0){
24   execl("/home/mozilla-1.9.1/firefox",
25   "/home/mozilla-1.9.1/firefox",
26   line, NULL);
27  }
28  else{
29   wait(seconds);
30   system(cmd2);
31   wait(1);
32  }
33 }
34 fclose (infile);
35 printf("Done!\n");
36 }
37 return 0;
38
39 }
```

Listing 3.2: Code for collecting string

3.2.4 Storage

We store collected strings on a DB if we want to analize them with interesting queries. More precisely, we used the following table:

```
create table jsstrings (
        id BIGINT NOT NULL AUTO_INCREMENT,
        place VARCHAR(10) NOT NULL,
        referrer LONGTEXT NOT NULL,
        script LONGTEXT NOT NULL,
        string LONGTEXT NOT NULL,
        date DATETIME NOT NULL,
        type VARCHAR(30) NOT NULL,
        PRIMARY KEY (id));
```

The field "`place`" is used to store the origin of the string (i.e., CONCAT, LOCAL, GLOBAL or PROP), the field "`type`" is used to distinguish good strings from malicious strings (it can assume three values: GOOD, MALICIOUS, UNDEFINED), and the other fields are self-explanatory. Then we have implemented the program in Listing 3.3 to save the strings on the DB.

This script uses as input a list as the example in Figure 3.1, and splits every line on the input file by the character "`\t`". Lines 10, 13-15 and 24 connect to the DB via JDBC. In line 33 we prepare the query to insert the string into the DB and lines 35-40 set the parameters of the query. The `PreparedStatement` class on line 34 is important to escape the characters that cause troubles to MySQL, and to preserve the shellcode if we want to analyze it afterwards.

3.2.5 Creation of baseline dataset with Google Safe Browsing

The lists used to create the malicious sets do not guarantee that all the sites were still active or malicious during the analysis of our tool. So, in order to make a more accurate analysis we chose to create a subset of malicious sites

```
1  String referrer,script,mystring,date,place,type;
2  if(args[1].equals("good")) type= args[1];
3  else if(args[1].equals("malicious")) type= args[1];
4      else type= "undefined";
5      BufferedReader bufferedReader = null;
6      bufferedReader = new BufferedReader(new
          FileReader(args[0]));
7      String line = null;
8      Connection c =
          DriverManager.getConnection(CONNECTION,p);
9      Pattern pat = Pattern.compile("\t");
10     while ((line = bufferedReader.readLine()) != null) {
11         String[] items =pat.split(line);
12         place=items[0];
13         referrer=items[1];
14         script=items[2];
15         mystring=items[3];
16         date=items[4];
17         String qry = "INSERT␣INTO␣jsstrings␣
              (place,referrer,script,string,date,type)␣
              VALUES␣(?,?,?,?,?,?);";
18         PreparedStatement pstmt=
              c.prepareStatement(qry);
19         pstmt.setString(1, place);
20         pstmt.setString(2, referrer);
21         pstmt.setString(3, script);
22         pstmt.setString(4, mystring);
23         pstmt.setString(5, date);
24         pstmt.setString(6, type);
25         pstmt.executeUpdate();
26         pstmt.close();
27     }
28 c.close();
```

Listing 3.3: Code for saving strings on MySQL database

that are flagged as malicious by Google Safe Browsing[6](from now on referred as GSB). To do this we created a Perl script that uses Google Safe Browsing API to filter the input sets of malicious sites creating a malicious subset of malicious sites.

[6]http://www.google.com/safebrowsing/diagnostic?site=google.com

```
1  use Net::Google::SafeBrowsing2;
2
3  open(FILE, "< malwarelist.txt");
4  open (MYFILE, "> data.txt");
5
6    my $gsb = Net::Google::SafeBrowsing2->new(
7      key      => "google apps key",
8    );
9
10 while ($riga = <FILE>)
11 {
12   $gsb->update();
13   my $match = $gsb->lookup(url => 'http://'.$riga);
14
15   if ($match eq MALWARE) {
16     print MYFILE $riga;
17   }
18 }
19
20 close (FILE);
21 close (MYFILE);
```

Listing 3.4: Code for filter the malicious set with GSB

The subset created by the program contains 4559 URLs, not a big number but sufficient to evaluate the features and to compare the results with the analysis of the other two sets. We chose to use both malicious set and GSB set because we are not certain that all the dangerous sites of the malicious set are flagged as malicious by GSB, but the sites that are present both on malicious set and GSB set have a very high probability to be malicious.

Chapter 4

Experimental Results

In this chapter we show the results of the evaluation of four features, with graphs and tables.

4.1 Feature 1: Distribution of Strings Length

Figure 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 show the results obtained.

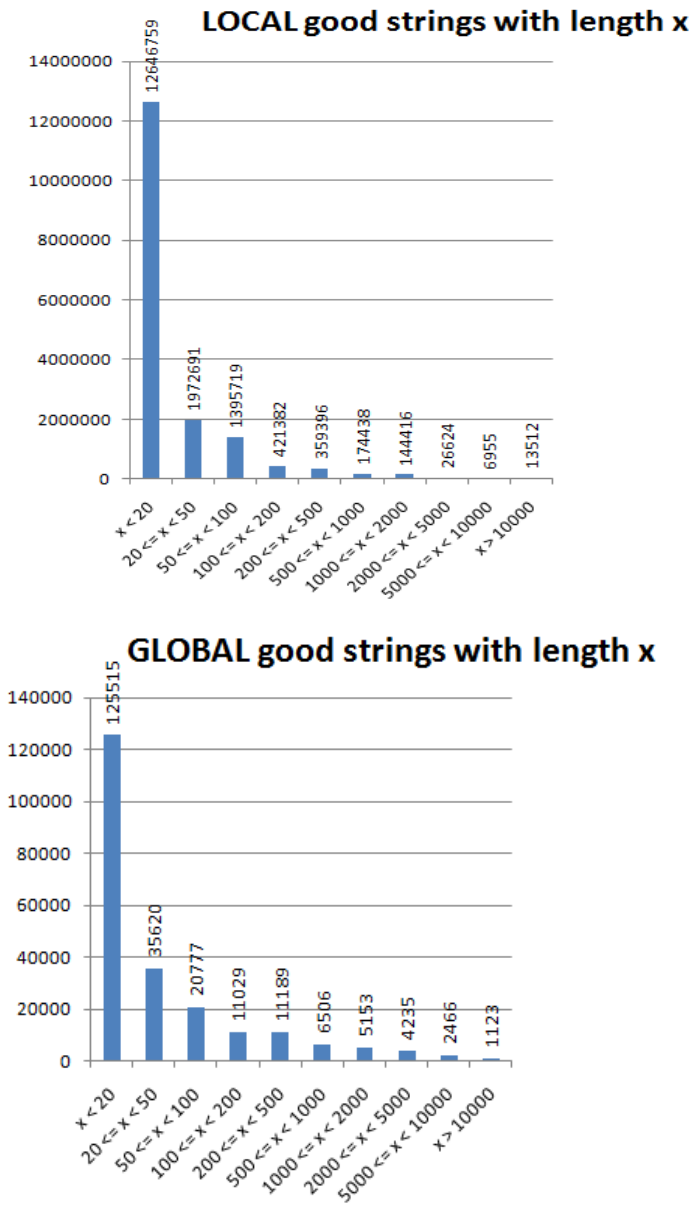

Figure 4.1: Cumulative distribution functions of the length of the local and global string allocated by good sites. Most of the strings have a length lesser than 20 and the number of strings decrease with the growth of the length.

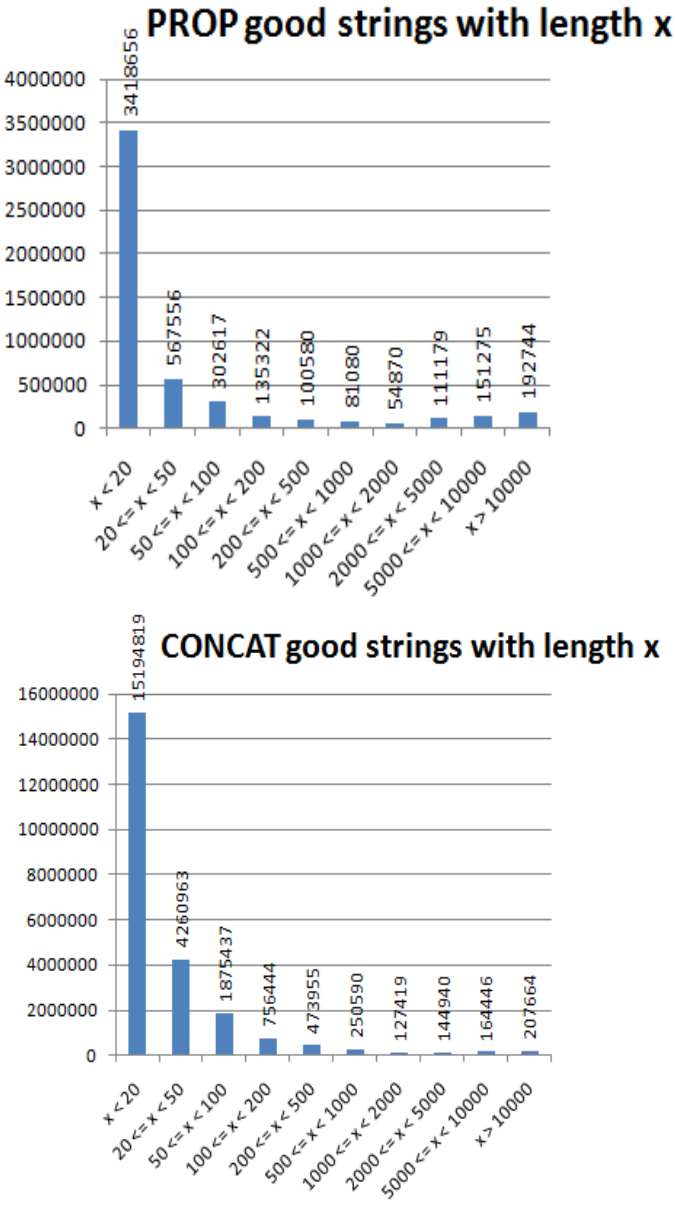

Figure 4.2: Cumulative distribution functions of the length of the property and concatenated string allocated by good sites. Most of the strings have a length lesser than 20 and the number of strings decrease up to 2000. After 2000 the number of string grows.

These are some consideration on the results:

- the majority of the strings in every histogram has a length that belong

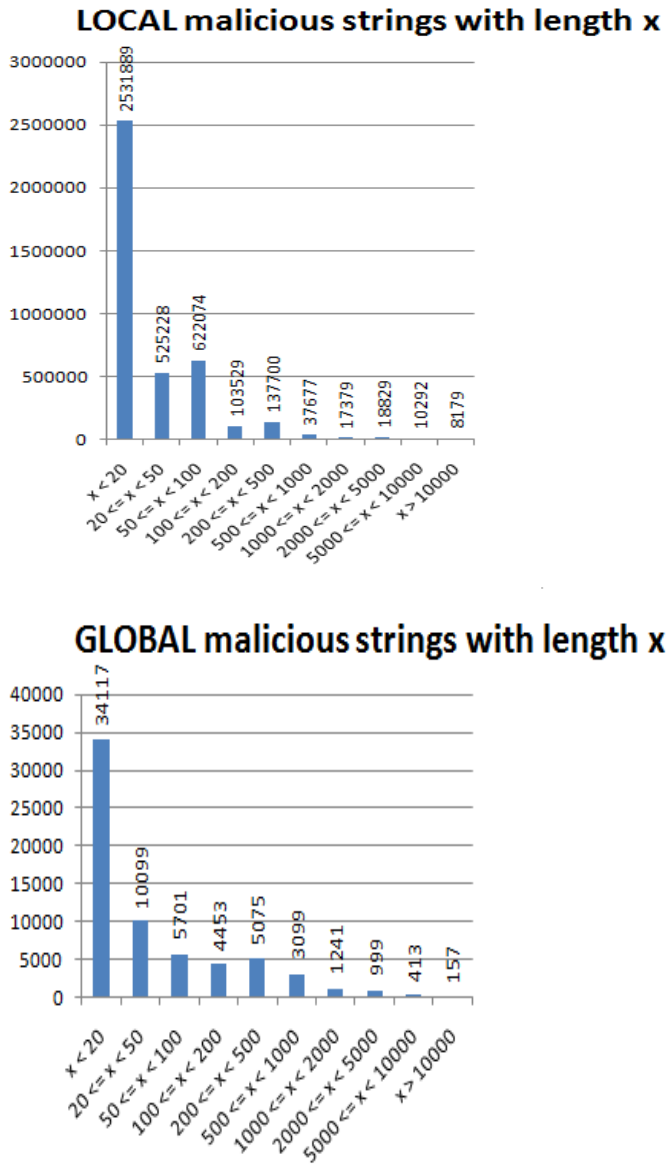

Figure 4.3: Cumulative distribution functions of the length of the local and global string allocated by malicious sites. Most of the local strings are concentrated in the interval 0-100, with few strings with a length greater than 100. Global strings histogram has a shape similar to the global histogram of good strings, with the exception of the interval 200-500 that shows a local maximum.

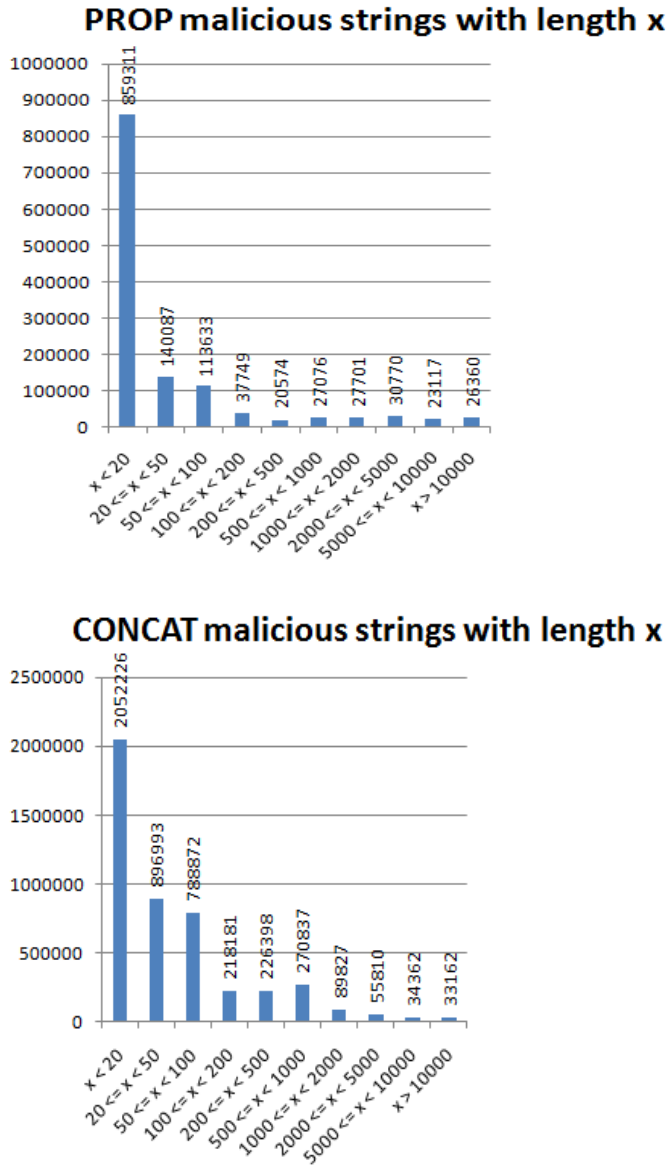

Figure 4.4: Cumulative distribution functions of the length of the property and concatenated string allocated by malicious sites. Most of the strings have a length lesser than 20. The histogram of the property strings shows that the number of string is regular for length greater than 100, whereas the concatenated histogram show 4 groups: strings with length smaller than 20, length between 20 and 100, length between 100 and 1000 and strings with length greater than 1000.

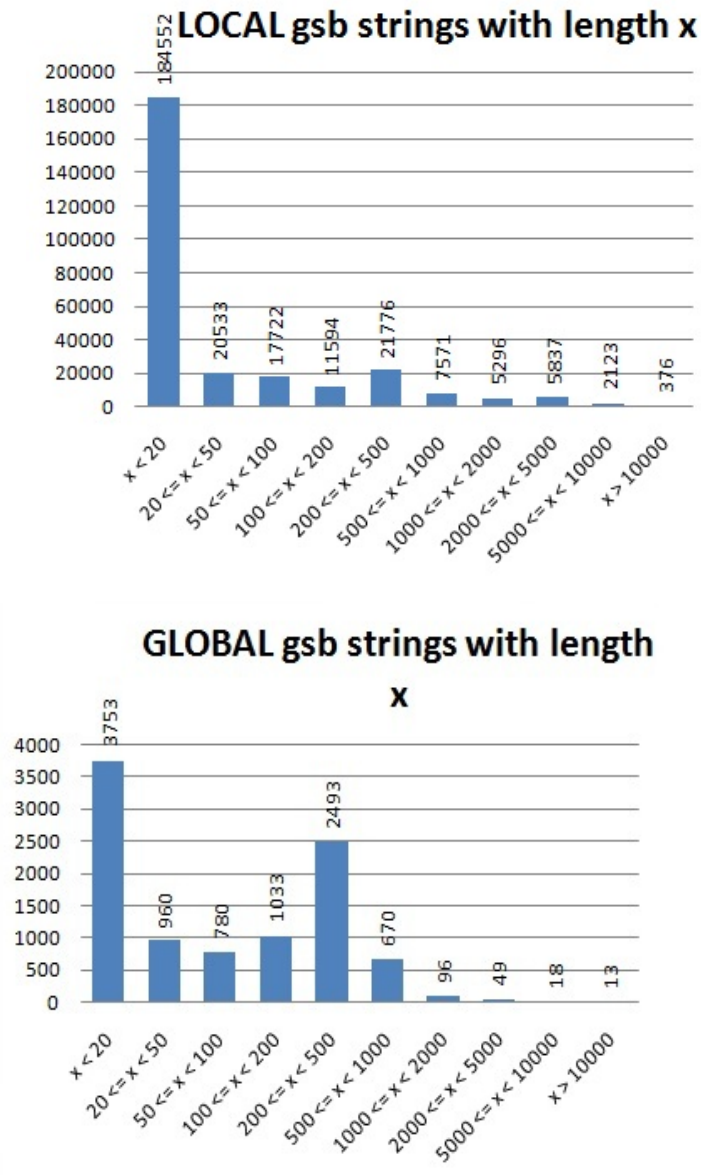

Figure 4.5: Cumulative distribution functions of the length of the local and global string allocated by GSB sites. Most of the strings have a length lesser than 20 and it is very interesting the local maximum of the global strings in the interval 200-500.

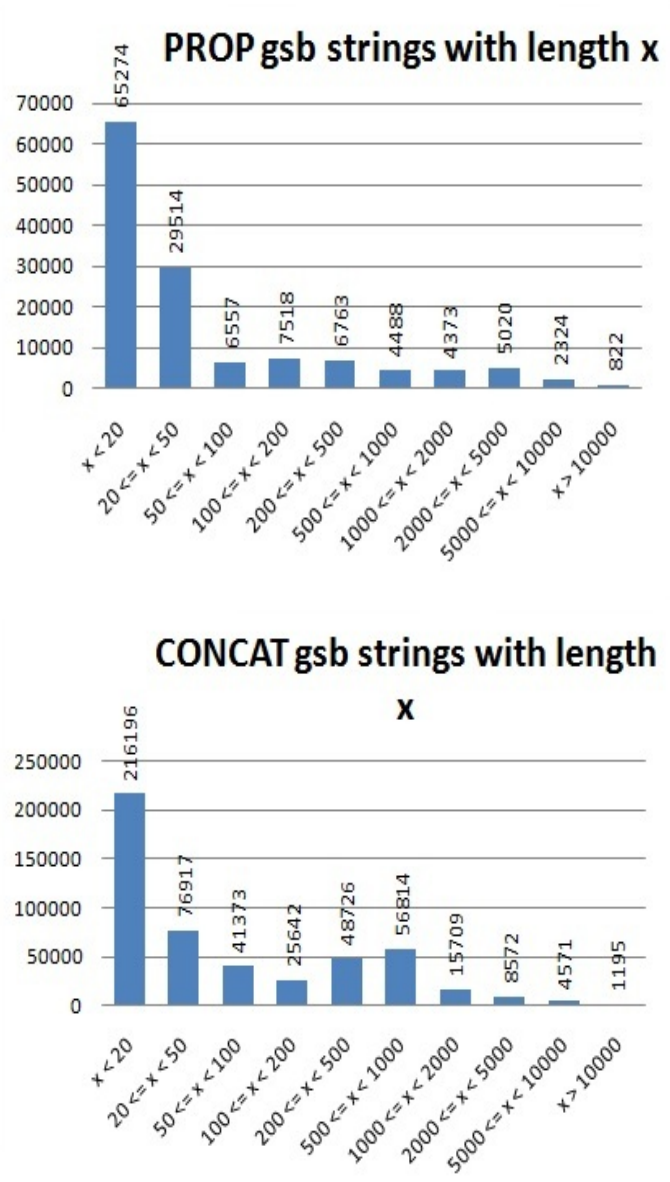

Figure 4.6: Cumulative distribution functions of the length of the property and concatenated string allocated by GSB sites. Most of the strings have a length lesser than 20 and the concat strings have a local maximum in the interval 500-1000.

to the interval 1-20. This means that usually JS variables have a short length or they are pieces of longer strings;

- all the distribution decrease with the growth of the length, with the exception of the strings that are property of an object (PROP) or concatenated in the good and malicious sets, and the global GSB variables. In particular, very long strings are results of a concatenation because it is easier for an attacker to avoid detection splitting the malicious machine code into smaller variables;

- the distribution of the global malicious variables and global GSB variables show a slightly different shape from the histogram of global good variables in the interval 200-500, with a local maximum in that interval;

- the distribution of the local malicious variables shows a shape similar to the one described in the previous point with respect to the histogram of local good variables in the interval 50-100.

Figures 4.7 and 4.8 compare the above results. We can see that malicious sites allocate more strings with a length of 50 to 100 characters as a result of a concatenation. We notice the same behavior also for local and property string. Also, concatenated strings allocated by good sites with a percentage higher than 2.5% have length lesser than 200, whereas the strings allocated by malicious sites with a percentage higher than 2.5% have length up to 1000. This is even more visible with the strings collected from the GSB set. We can conclude that attackers use more concatenated strings as a vector to launch an attack.

It is also very interesting that malicious sites allocate more long string that good sites. This confirm the fact the machine code used to launch attacks needs to be stored inside long strings, otherwise the attacks have no effects on the victim machines.

In summary:

- malicious sites allocate more local strings with length of 20-100 and global string with length of 100-1000. This is because usually the

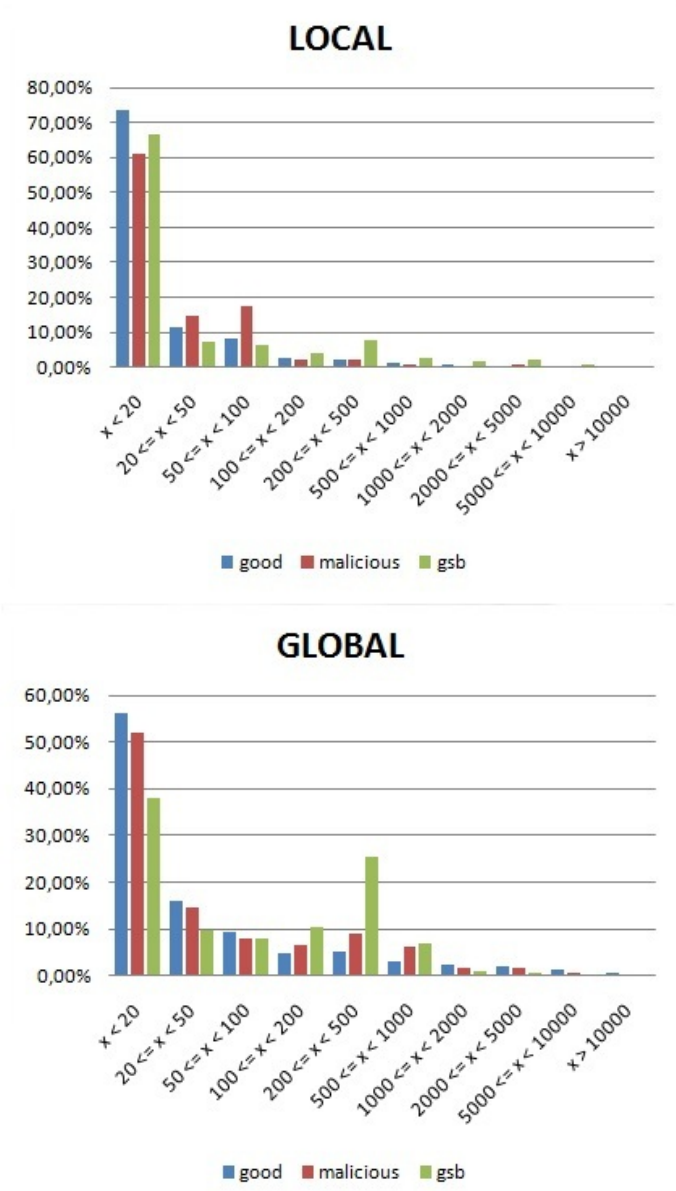

Figure 4.7: Comparation of local histograms and global histograms. Malicious sites allocate more local strings with length in the interval 20-100 and global string with length in the interval 100-1000. Other intervals show no big differences.

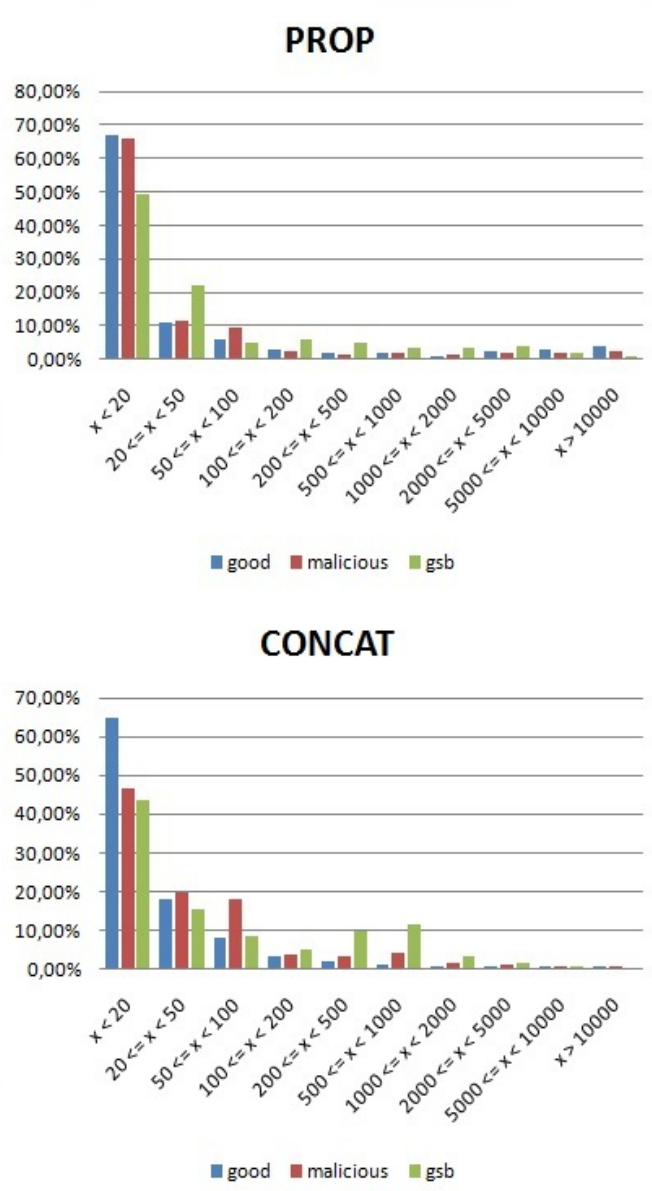

Figure 4.8: Comparation of property histograms and concatenated hystrograms. Malicious sites allocate more strings that are result of a concatenation with length in the interval 50-2000. In the property histogram there are differences only in the interval 50-100.

strings that contains malicious code are splitted into smaller pieces to evade detection.

- malicious sites allocate more strings that are result of a concatenation with length of 50-2000. In fact concatenation is used more by malicious users to reconstruct the original string that contains machine code. This is done because machine code must be allocated in a single string to be executed.

4.2 Feature 2: Presence of Obfuscation or Code Generating Functions

We hereby concentrate on the presence of functions used for obfuscation of code or for dynamic generation of code. The results are shown in Figure 4.9.

The most interesting words are *unescape(*, *concat(* and *document.write(*. In fact these words are more frequent in malicious strings than in good strings because *unescape(* is used inside functions for obfuscation, whereas *concat(* and *document.write(* are used inside functions for dynamic generation. In particular *concat* function is used to built dangerous strings made by smaller and harmless strings. Other interesting words are *eval(*, *setTimeout(* and *document.createElement(* because their presence is very high inside good strings even if they could be used very easily for malicious purpose. The evaluation of this feature on the GSB set confirm our intuition on the functions *unescape(*, *document.write(* and also on *document.createElement(* and *substring*. In fact these two functions are used for splitting the strings and for dynamic generating code to avoid security tools, and in this particular subset is evident that are useful to distinguish the behavior of a malicious page.

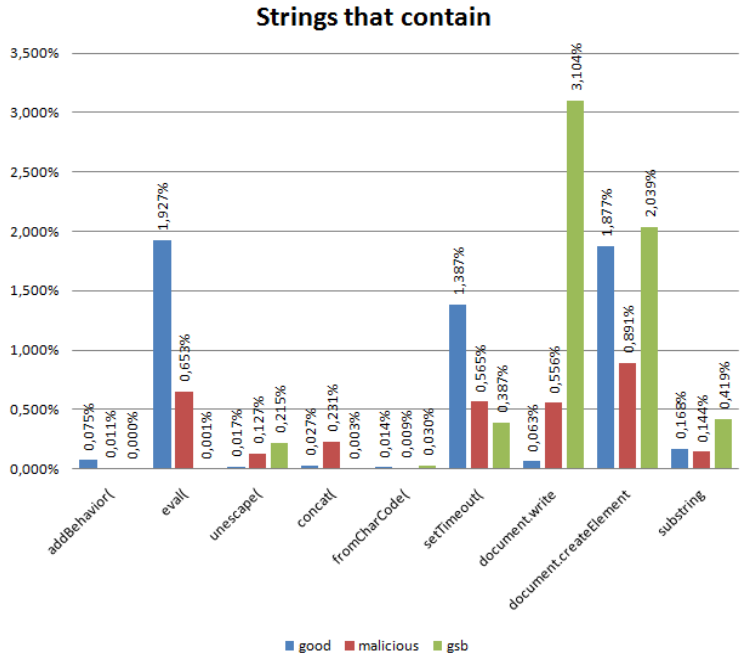

Figure 4.9: Analysis of substrings to count the number of occurrences of functions used for obfuscation or dynamic generation of code. The words more used by malicious sites are unescape (obfuscation), document.write (dynamic generation) and substring.

4.3 Feature 3: Count of Reserved Words

In Section 4.2 we focused on the presence of some words that represent functions. In this section we focus on the presence of reserved words of the JS language[1]. The results are summarized in Figure 4.10, 4.11 and 4.12, showing only the most frequent words. The histograms show the percentage of variables that contain reserved words with respect to the total of variables collected. We can see that words like *unescape* are also inside these figures, but *concat* and *document.write* are not present because they are not reserved JS words. The function *unescape* and the word *Hidden* are very frequent in malicious strings because are used for (de)obfuscation. Another interesting word is *Form*. Even if it is very frequent inside good pages, there are more occurrences inside malicious variables because many often the attackers put malicious code inside hidden forms that are not visible to the victims. The evaluation of this feature on the GSB set confirm all previous results, with the exception of the word *Hidden*, that have a lesser occurrence with respect to good variable. However, we recall that GSB set is smaller than the other set, and so in this particular case the word *Hidden* has a lesser frequence (about 5.5%).

[1] `http://www.quackit.com/javascript/javascript\_reserved\_words.cfm/`

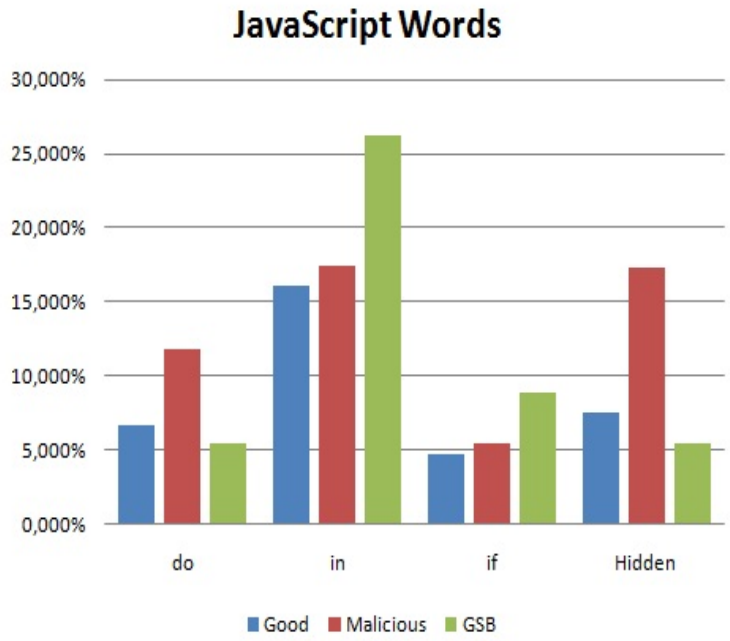

Figure 4.10: Analysis of substrings to count the number of occurrences of JS reserved words. The most interesting word in this histogram is Hidden (used inside obfuscating functions).

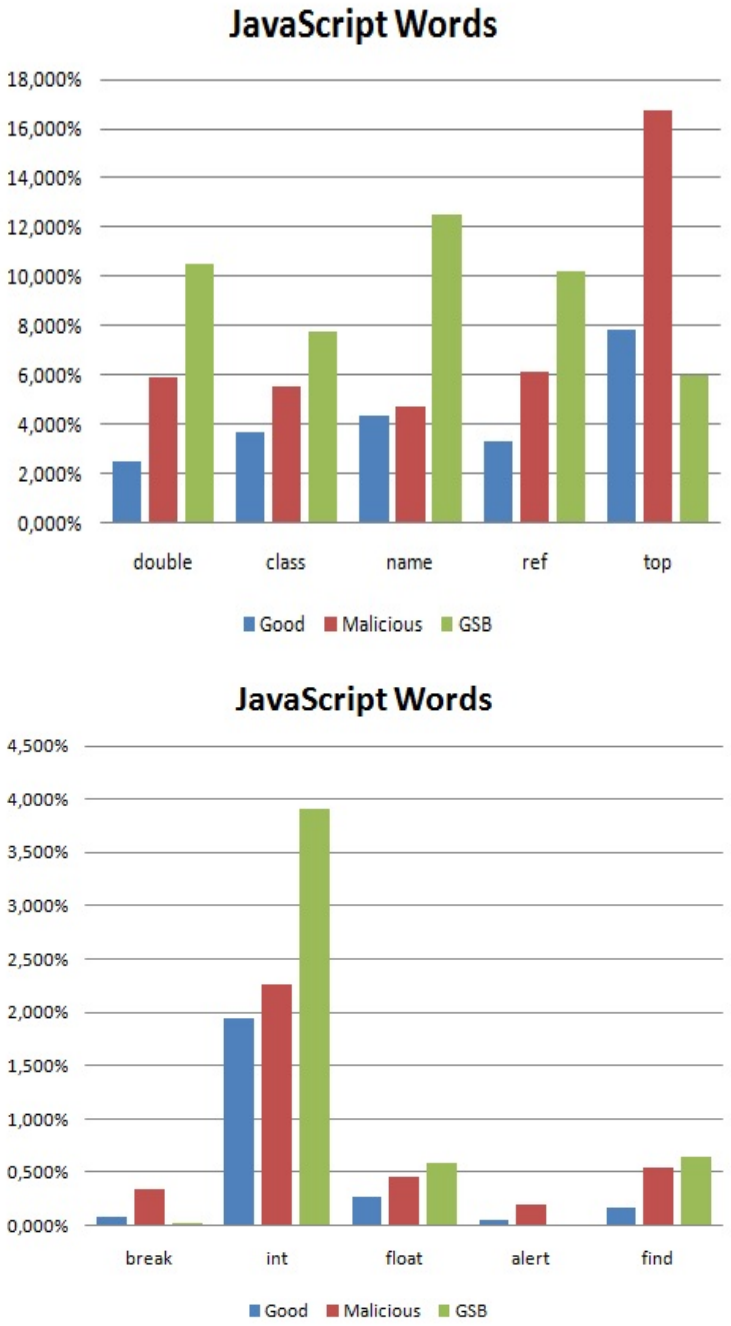

Figure 4.11: Analysis of substrings to count the number of occurrences of JS reserved words. These words are not associated with specific functions but could be used to cast a decision on the maliciousness of a web page.

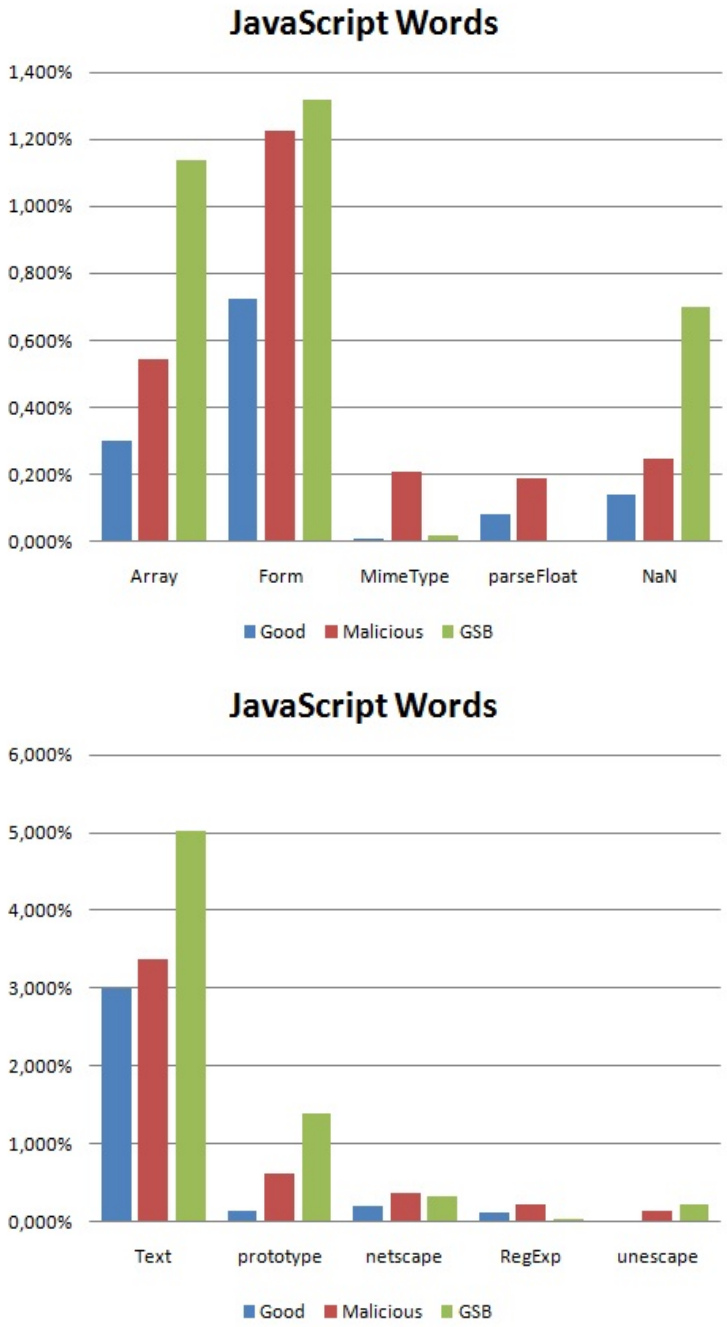

Figure 4.12: Analysis of substrings to count the number of occurrences of JS reserved words. The word unescape, used inside functions for obfuscation, is present more inside variables allocated by malicious sites.

4.4 Feature 4: Ratio of the number of collected variables to the number of distinct referrers

The results of the evaluation are shown in Figure 4.13. The histogram shows that the ratio of collected variables to distinct referrers evaluated on good set is bigger than the ratio on malicious and GSB sets. This means that many good JS variable share the same referrer, reminding that we have collected 45,958,061 variables from the good set, 10,051,176 variables from the malicious set and 915,613 variables from the GSB set. Our opinion on this result is that good sites allocates more variables with respect to malicious sites because of the advanced functionalities they offers. In fact the majority of malicious sites shows to the victim users only small messages or popups.

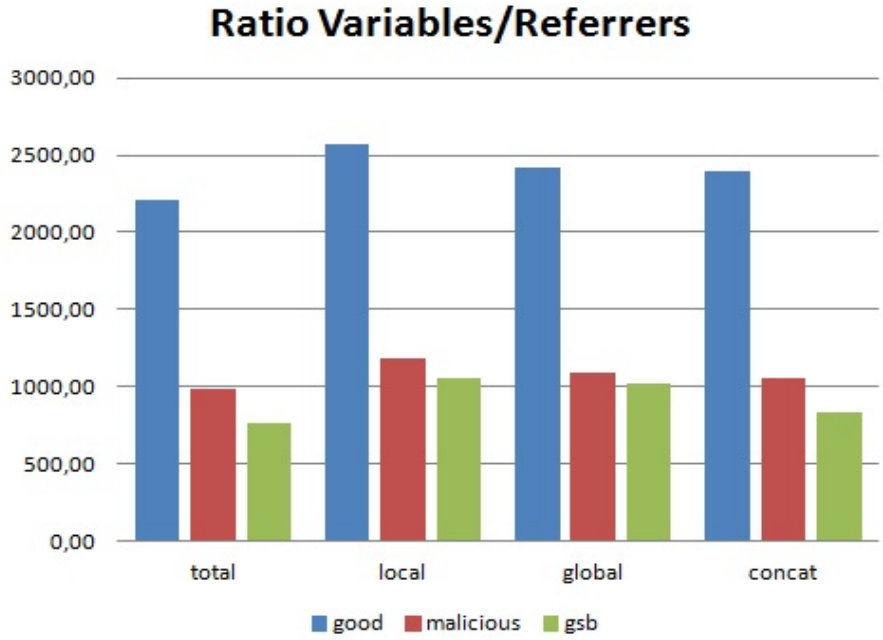

Figure 4.13: Number of collected variables divided by the number of distinct referrers. The figure show that good strings have a lesser percentage with respect to malicious sites. This means that every good referrer allocates more strings than malicious referrers.

4.5 Other Features

During the preparation of this thesis we have evaluated some features that gave similar results on both good and malicious sites and so they were not very useful. Some examples of these features are: we evaluated how the collected good strings and the malicious strings are composed, by creating two pie charts with the percentage of the place of capture (local, global, property or concat) of the strings. Figure 4.14 shows the pie chart relative to the good strings and the malicious strings. The tables display other parameters extracted like average length of the strings and the string with max length (probably outliers). The differences between the two charts are no evident, and so we cannot use this feature alone to recognize a malicious site.

Tables 4.1, 4.2 and 4.3 show some parameter evaluated from the collected strings. The strings with highest length are composed by JavaScript code, and so it is better to analyze the words they contain instead of other parameters. Of course the average length of the strings is very high because of the outliers with an enormous length (e.g. over 800000 characters for local, property and concatenated good strings).

Another feature we have analyze that was not helpful for our purpose was the number of variables that contains only alphanumerical characters. Figure 4.15 shows that the percentage of the variables only composed by alphanumerical characters were very similar between malicious and good sites. The variables collected from GSB set show some differences, but this is not sufficient to be used for the goals of this thesis work.

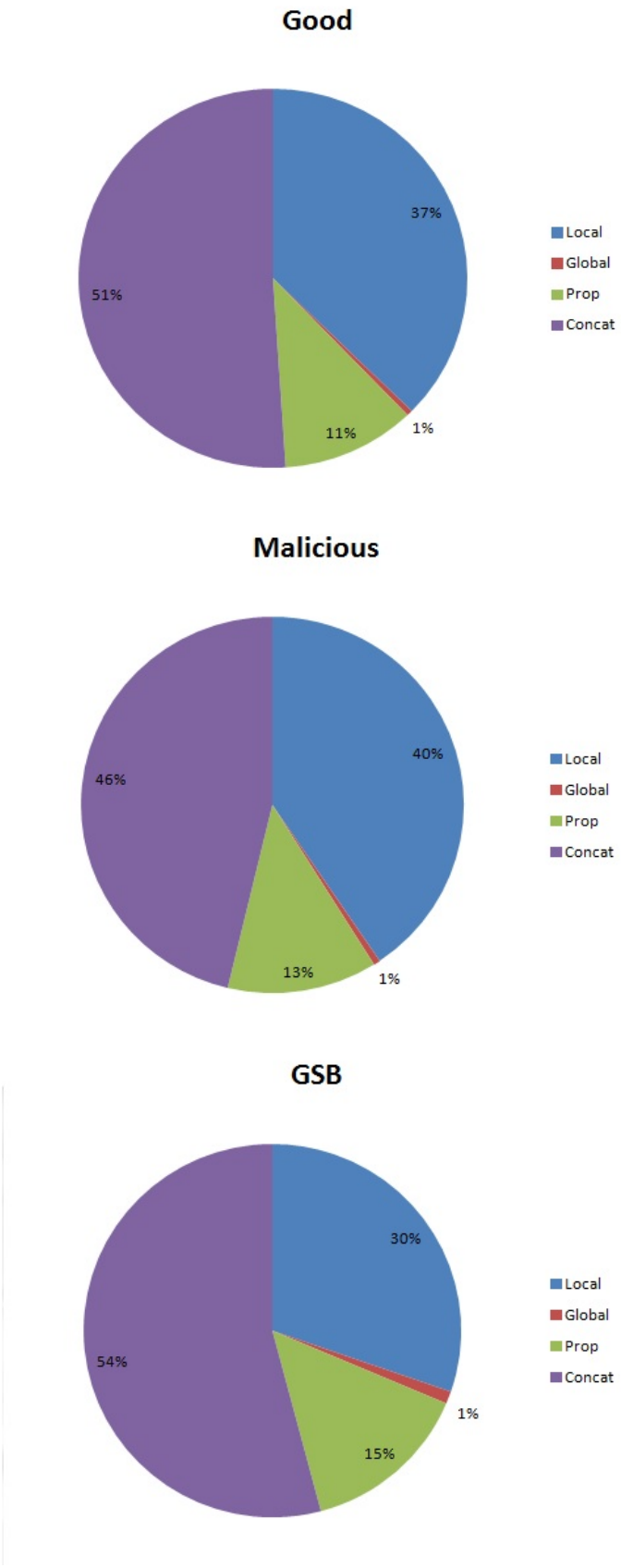

Figure 4.14: Comparison between the composition of good strings and malicious strings grouped by their origin. The differences are not helpful to cast a decision on the behavior of a web page.

Place	Good N	Good Avg(length)	Good Max(length)
Local	17161892	73,0796	800693
Global	223613	318,5576	344766
Prop	5115879	884,6173	800696
Concat	23456677	250,0295	800693

Table 4.1: This table shows the evaluations of some parameters such as average and maximum length of the strings and the number of the good strings collected.

Place	Malicious N	Malicious Avg(length)	Malicious Max(length)
Local	4012776	116,9695	336693
Global	65354	250,7348	344766
Prop	1306378	564,2842	288651
Concat	4666668	310,6884	521914

Table 4.2: This table shows the evaluations of some parameters such as average and maximum length of the strings and the number of the malicious strings collected.

Place	GSB N	GSB Avg(length)	GSB Max(length)
Local	277380	237,5896	115331
Global	9865	270,3812	521901
Prop	132653	431,695	38155
Concat	495715	329,5075	521902

Table 4.3: This table shows the evaluations of some parameters such as average and maximum length of the strings and the number of the GSB strings collected.

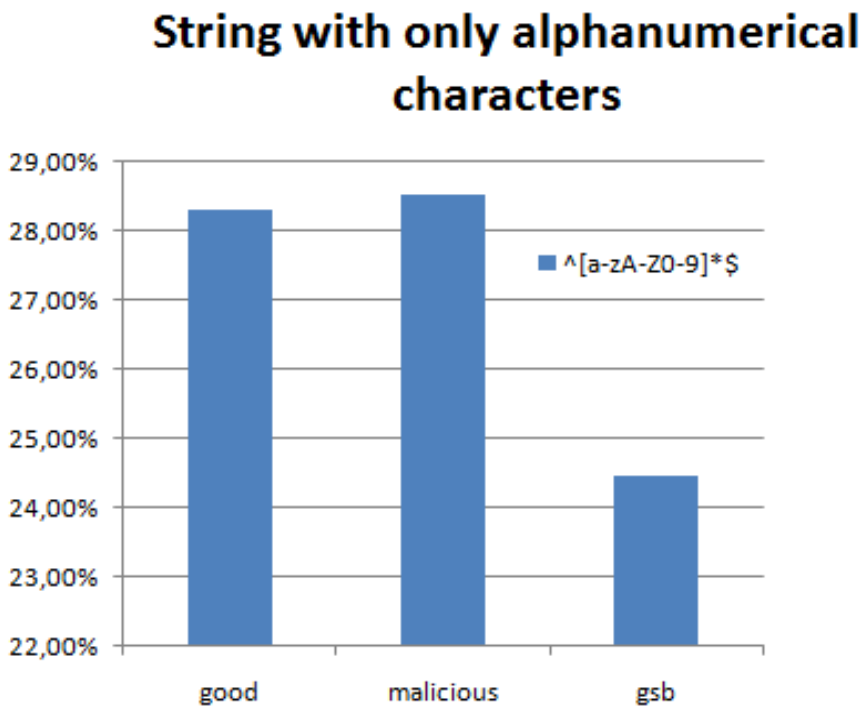

Figure 4.15: Analysis of characters inside the allocated strings. The histograms are very similar and we cannot use this feature for the goals of our work.

Chapter 5

Conclusions

In this work we evaluated four features that characterize string variables allocated by JavaScript code found in malicious web pages. We found that good JS code allocates more variables with respect to malicious JS code and the majority of those variables has length lesser than 20. In addition, an higher percentage of malicious variables are originated by concatenation functions. Plus, variables allocated by malicious JS code contains words and functions used for (de)obfuscation and dynamic generation such as `Hidden`, `concat`, `unescape` and `document.write`. We have also found features that do not show differences between malicious and good sites, such as the composition of the collected strings and the number of variables that contains only alphanumerical characters.
During the preparation of this work we made some choises that can be improved in future updates.

For example it could be convenient to save the strings directly into the DB from the code added into Spidermonkey, to save a lot of time. In this way it is possible to open multiple pages on the browser, instead that one at time, without creating writing overlaps on the output file.
Another possible upgrade is to modify the program the checks the list of URLs with the modified browser. In our case we wait 12 seconds before closing the browser's window, and this value is too big for some sites (e.g., sites hosted near the client or lightweight sites), while is too small for other

sites (e.g., sites hosted far from the client or heavyweight sites). So it could be better to change the waiting function in such a way that the program closes the browser's window when the checked page is completely loaded. Another improvement could be to open many tabs on the same window, with a different URL on each tab. Finally, like we said in the previous section, there are a lot possible evaluations that we can do on our data. We can make other analysis on the characters of the strings, to verify if they are part of executable shellcode, or if the strings contains other specific words, used for new attacks. For a more precise results is could be possible to use for malicious strings collection only the URLs flagged as malicious by Google Safe Browsing [1]. In this way those sites contain their malicious code and so we are sure that they are not cleaned or closed.

[1] http://www.google.com/safebrowsing/diagnostic?site=www.example.it

Bibliography

[1] Spidermonkey (javascript-c) engine. www.mozilla.org/js/spidermonkey/.

[2] Davide Canali, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *Proceedings of the International World Wide Web Conference (WWW)*, Hyderabad, India, March 2011.

[3] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the World Wide Web Conference (WWW)*, Raleigh, NC, April 2010.

[4] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, SteveBeattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard:automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, January 1998.

[5] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Low-overhead mostly static javascript malware detection. Technical report, Microsoft Research Technical Report, November 2010.

[6] Manuel Egele, Engin Kirda, and Christopher Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *INetSec 2009-Open Research Problems in Network Security*, April 2009.

[7] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Detection of In-*

trusions and Malware and Vulnerability Assessment, Milan, Italy, July 2009.

[8] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from Monkey Island: Evading High-Interaction Honeyclients. In *Proceedings of Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Amsterdam, The Netherlands, July 2011.