# Research Project 2: Metasploit-able Honeypots

Wouter Katz

July 5, 2013

**Supervisors**
Jop van der Lelie (NCSC-NL), Bart Roos (NCSC-NL)

# Abstract

News of computer systems being hacked has become so common that it no longer raises eyebrows. As more and more systems around us get Internet connectivity (TV, mobile phone, even cars) and hacking tools are freely available and easy to use, the need to gain more insight into the activities of attackers of computer systems is high.

One such method to gain insight into the activities and methods of hackers are so-called honeypots. These systems are built to attract hackers, and log all their activities. However, most honeypot software contains outdated vulnerabilities and do not provide insight into which exploit is used by the attacker, but require manual analysis of the saved data by an administrator.

This research focuses on automating the process of extracting exploit signatures from network traffic, so that these signatures can be implemented in honeypots. The methods described in this research allow for signatures of new exploits to be easily extracted and automatically detecting these signatures occurring in traffic towards the honeypot. This gives honeypots a better view on the exact exploits used by attackers while requiring significantly less maintenance to get this information.

# Contents

# 1        Introduction

Many of the free honeypot software packages available are not being actively maintained anymore. This means that the vulnerabilities emulated within these honeypot software packages are often outdated, and give a skewed vision on threat detection and assessment. This research is aimed at finding an automated method to recognize and detect exploits based on their network traffic, allowing for emulation of newer vulnerabilities within honeypots.

## 1.1 Research Question

The main research question for this report was the following:

***How feasible is an automated method to detect specific exploits on a honeypot by monitoring network traffic of exploits?***

This question can be split up in the following sub-questions:

- What setup is needed in order to have exploits successfully complete their exploit against a honeypot?

- What is the best method to process network traffic to/from the honeypot to extract and match a unique signature from exploit traffic?

- How successful is this method?

## 1.2 Previous Work

The exploitation mechanism and the most used methods of exploitation (stack overflow, heap overflow and format string attacks) have been described in detail [1–4]. The use and development of shellcode are well documented [5,6], and show how shellcode started out as a simple, static part of the exploiting process. This form of shellcode is easy to detect by monitoring network traffic for the static patterns occurring within the shellcode, and research on this has been done in detail [7, 8].

Attackers then evolved their methods to avoid detection, both by upgrading their exploiting techniques [9, 10], and by advances in payloads used. Static payload signatures were replaced by polymorphic payloads which are encrypted or obfuscated, making detection by signatures in network traffic a lot more difficult [11, 12]. Solutions for this have been found by emulating execution of the shellcode in order to classify it as malicious or benign code [13, 14].

Honeypot systems have been used to attract attackers and log malicious activity against emulated vulnerable services [15, 16]. Most honeypots merely log details of activities on the honeypot system, although some honeypot systems are advanced in a way that they emulate execution of shellcode [17, 18], or present attackers with a fake shell on an emulated system in order to monitor their activities [19, 20].

A method to automatically create honeypot scripts to emulate traffic has been researched by [21], however this method only focuses on creating a method to reverse engineer a protocol and automatically create a honeypot script to emulate this protocol, not to emulate specific vulnerabilities. Automatically creating signatures for Network Intrusion Detection Systems from sources such as exploit traffic has been researched [22], but this only focuses on creating signatures and not on matching these signatures against live traffic.

# 2

# Terminology

This chapter covers some of the terminology that is used throughout this report.

## 2.1 Exploit

According to [23], exploiting a vulnerability can be defined as taking advantage of a flaw in a system's security that can lead to an attacker gaining access privileges on an unintended level.

An exploit usually consists of two parts. One part is the payload, the actual machine-code to be executed on the target system. This payload can spawn a new process, create a remote shell on the target system, and many other possibilities. The second part of the exploit is a sequence of data and/or commands which triggers the vulnerable application to execute the payload.

## 2.2 Payload Encoding

Payloads are a static piece of code which can be plugged into any exploit, given that technical preconditions are met, such as the available amount of space for a payload is not exceeded, or the payload does not contain any characters that are not correctly interpreted by the application that is being exploited. Payloads being static makes them a good target for signature based detection in network traffic. Many Network Intrusion Detection Systems (NIDS) use this approach to detect commmon patterns occurring within payloads.

To avoid this form of detection, encoding of payloads is used. Encoding of a payload can range from simple operations such as translating all letters to uppercase to more sophisticated operations like XOR encryption of the payload. The latter method results in a payload containing a small decoding routine stub prepended to the encrypted shellcode, as can be seen in Figure 2.2. Upon execution the decoding stub decrypts the payload so it can be executed in its unencrypted form. XOR-based encryption using randomly generated encryption keys is an excellent method of hiding the original shellcode, but the decoding stub itself can still easily be recognized by signature-based detection. Polymorphic shellcode is an even more advanced method to avoid detection, using multiple levels of obfuscation. Besides having the original shellcode obfuscated, other methods to avoid detection are also used, such as randomly replacing individual instruction(s) that make up the decoding routine with other instructions which provide the same result as the replaced instruction(s). This makes the decoding

stub a lot more difficult to detect using signature-based detection.
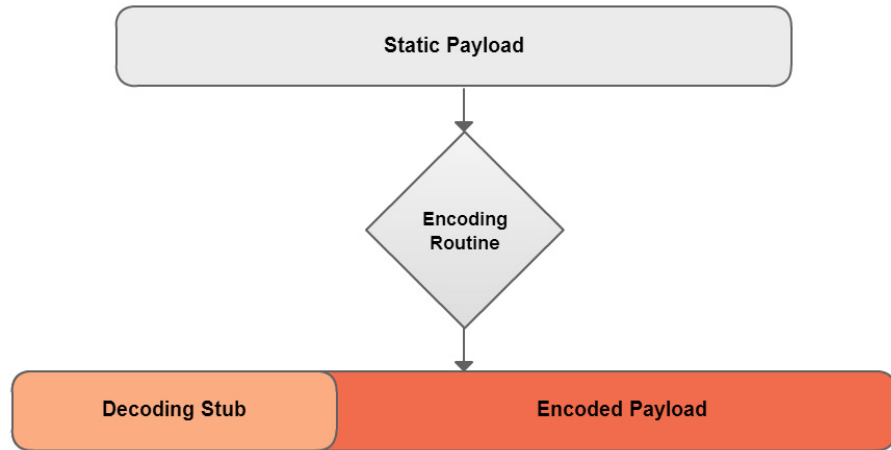
## 2.3 Honeypots

An option to gain more insight into the often hidden activities by attackers and their methods, is to attract attackers and watch from the sideline how they operate. Honeypots are a prime example of doing so. A good definition on what honeypots are is given by [24]: *"A honeypot is used in the area of computer and Internet security. It is a resource which is intended to be attacked and compromised to gain more information about the attacker and the used tools."*

To attract attackers, honeypots emulate vulnerabilities which attackers believe can be abused, while in reality the system is not vulnerable and monitors all attacker activity. Emulated vulnerabilities can range from open mail relay systems which attackers believe they can use to send out spam, to emulating software versions which are known to be vulnerable to remote code execution.

Since the sole purpose for honeypots is to attract attackers, one can presume that any activity towards the honeypot is malicious.

Honeypots can be categorized as either low or high interaction. Low interaction honeypots merely emulate vulnerabilities, and thus prevent the system running the honeypot from actually being compromised. High interaction honeypots often run the actual vulnerable software, and require a reinstall or reimaging of the OS after a compromise to restore the honeypot system to its original, vulnerable state.

# 3 Process

The main goal of this project is to extract signatures from the network traffic generated by executing exploits against a honeypot. If a unique signature can be generated per exploit, the honeypot can not only emulate the vulnerability, but also recognize which exploit was used. The method to do this needs to be as general as possible, so that it can work on any protocol.

This chapter will describe in general the process from firing exploits against the honeypot to detecting exploits based on extracted signatures, as well as theoretical background on which each step of the process is based.

A flow diagram of the process is shown in Figure 3.1.

Figure 3.1: Flow diagram of the process used in this research
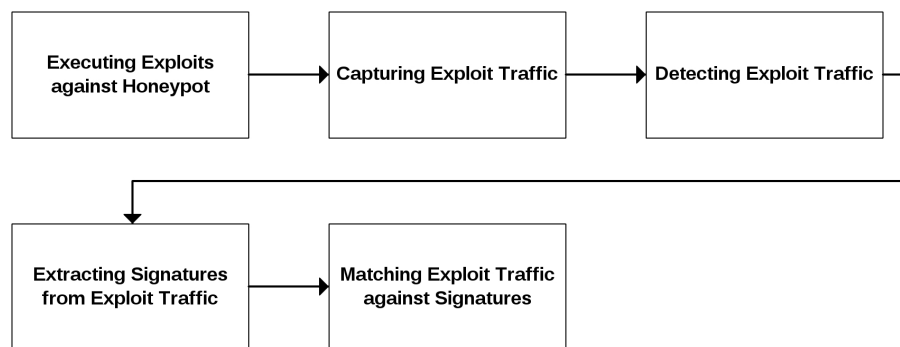


## 3.1 Capturing Exploit Traffic

When deployed in a real-life scenario, all attack attempts against the honeypot system will originate from the Internet, meaning that the traffic between the honeypot system and the attacker's system will contain all information needed to extract and match for exploit signatures. For this reason, all traffic has to be saved for analysis. With the captured traffic, all stages of exploitation can be looked into to find similarities in exploit traffic, save payloads, and perhaps even find new exploits.

The de facto method of capturing network traffic is to use the tcpdump[1] utility. This utility allows for capturing of any traffic that comes through a network interface, and has advanced capabilities when it comes to filtering and storing of network traffic in

---

[1] tcpdump: http://www.tcpdump.org/

Packet Capture (PCAP) format. However, tcpdump is not ideal for matching exploit traffic due to the fact that tcpdump has to be stopped to parse the output file that tcpdump produced, which would mean that any traffic occurring while tcpdump is not running, would not be captured. Another reason for not using tcpdump is that looking for a specific network flow can get tedious using tcpdump or PCAP libraries, while a custom solution might be more fitting for this.

## 3.2 Detecting Exploit Traffic

An exploit usually follows a fixed process of first setting the stage for the exploitation process (e.g. by logging in first), and then proceeds to do the actual exploiting. Since this model takes place over multiple flows, each network flow to/from the honeypot should be available for analysis separately. To keep an overview on all separate flows, all flows will be grouped by the connection within they occur.

These flows should be analyzed to detect anomalies within the traffic that can distinguish exploit traffic from normal traffic, so that flows containing exploit traffic can be marked as suspicious for further analysis.

## 3.3 Extracting Signatures from Exploit Traffic

To extract signatures from the suspicious flows several approaches were considered.

The first option is to look for static patterns in the suspicious flows. This means looking at multiple traffic captures for the same exploit, and finding strings in the traffic that all traffic captures have in common. To accomplish this, the Longest Common Substring (LCS) algorithm can be used, which locates the longest string which is present in all input strings. The following pseudo-code explains the inner workings of an implementation of LCS:

```
1   // input_data = array of input strings
2   function lcs(input_data):
3           longest_substring = ""
4           if size of input_data > 0 and length of each input_data > 0:
5                   for (i = 0; i < length of input_data[0]; i++):
6                           for (j = 0; j < length of input_data[0] − i + 1; j++):
7                                   if j > length of longest_substring and ←
8                                           substring(input_data[0], i, i+j) in all input_data:
9                                                   longest_substring = substring(input_data[0], i, i+j)
10          return longest_substring
```

The second approach is to use learning algorithms for classification of the network traffic. By using for example Support Vector Machines (SVMs) or Bayesian statistics, supervised machine learning could be trained and used to classify exploit traffic.

Machine learning algorithms work by extracting so-called features from input data. These features are measurable numeric properties of the input data. For each input data, features are combined into a feature vector, which is used by the machine learning algorithm to classify the data by means of mathematical operations on the feature vector.

Which algorithm should be chosen for this research depends on the input data (i.e. the traffic flows).

The extracted signatures will be stored in a database, allowing for easy adding/removing/altering of signatures.

## 3.4 Matching Exploit Traffic against Signatures

When signatures are generated, the honeypot has to be able to match incoming traffic against these signatures to detect a possible exploit based on its signature. This should be done on-the-fly as the traffic comes in, to make detection results available instantly, and not having to process all saved network traffic afterwards to detect exploits.

# 4 Approach and Methods

This chapter describes the various steps that are taken to set up the required software and means to both capture and analyze the traffic. Any issues that are encountered during this process are described, as well as possible solutions to overcome or circumvent these problems.

## 4.1 Setting up Testing Environment

In order to conduct the analysis in a suitable environment, a setup is needed in which only traffic takes place that is part of the experiment, and unwanted traffic from the Internet does not reach the test setup. To accomplish this, two separate virtual machines are set up, both running Debian GNU/Linux release 7.0 (codename Wheezy). Both virtual machines are connected to the Internet by means of bridged networking. The bridged network connections are firewalled, so no traffic from the Internet can reach the virtual machines. This setup is visualized in Figure 4.1.



Figure 4.1: Setup of the testing environment
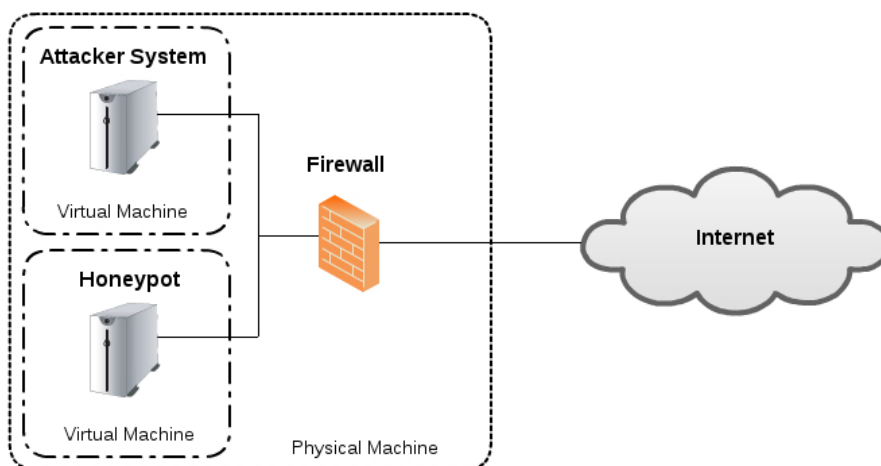
The first virtual machine will be set up as attacker machine, and a number of exploits will be placed to attack the honeypot virtual machine. The second virtual machine will be set up as the honeypot machine.

Using this testing environment setup, on the attacker virtual machine Metasploit Com-

munity Edition version 4.6.2-1[1] is installed. Metasploit is a very mature penetration testing software, containing over a thousand different exploits and several hundred payloads. Metasploit has a modular set up regarding to its manner of exploiting, allowing a user to choose an exploit, choose a payload to execute on the target system, and choose an encoder to encode the payload with. This, combined with its ease of use, makes Metasploit a popular tool among hackers as well. For this reason, Metasploit exploits were chosen to create honeypot software around.

Given the wide variety of exploit categories within Metasploit, focus is put on only the exploits within Metasploit that target File Transfer Protocol (FTP) servers. The FTP server exploits are chosen because they make up for one of the largest subsets of exploits that use 7-bit ASCII for communication within Metasploit, using a protocol with very little overhead. Starting off with a protocol that does all communication in ASCII makes development, debugging and analyzing of traffic, methods and algorithms substantially more effective than when dealing with more sophisticated protocols due to the ability to manually inspect traffic without having to decode it.

From all the exploits within Metasploit that target FTP servers, exploits which abuse vulnerabilities such as directory traversal or the ability to write anywhere on the target system are not included in the research. These exploits allow users to upload/download files from the target system, and do not include payloads in their traffic. Creating a vulnerable honeypot script to emulate vulnerabilities abused by these exploits would require substantial effort, such as emulating listings of popular directories which would be used by attackers to write their files to (e.g. C:\Windows\System32), since empty directories would raise suspicion from attackers. For this reason, only exploits that send a payload are included in the research. In total, 37 FTP server exploits within Metasploit fit this criterion, and are used as the exploit test set for this report. These exploits can be found in Appendix A.

On the honeypot virtual machine, Honeyd was initially chosen as honeypot software. Honeyd provides a set of scripts emulating specific services, FTP being one of them. The FTP emulating script within Honeyd is very limited, and merely returns the correct response to a limited number of FTP commands as specified in RFC959 [25].

When testing, it became clear that some Metasploit FTP server exploits did not send their payload to the Honeyd FTP script. There are two reasons for this:

- Some Metasploit exploits check the FTP software name and version number when connecting to the Honeyd FTP script. If this does not match with what the exploit expects to see, the exploit aborts.

- Some exploits use FTP commands that were not emulated by the Honeyd FTP script. For some of the Metasploit FTP exploits, the response from the Honeyd FTP script is checked, and if a certain FTP command that is used by the exploit is not implemented in the Honeyd FTP script, the exploit aborts due to an unexpected response.

To overcome these shortcomings, a custom FTP server emulator was written in Python for this research project. This script contains a number of features to aid the research:

- Additional FTP commands implemented;

- Using an external Python file to be used as database for FTP server banners;

- Being able to switch FTP server banners by providing a command-line argument, or serving random FTP server banners for each connection;

- Saving of exploit traffic;

- Flagging suspicious traffic.

To obtain a list of FTP server banners, each Metasploit FTP server exploit is examined to see which specific FTP software name and version number the exploit targets. With

---

[1]  Metasploit - Penetration Testing Software: http://www.metasploit.com/

this list, the Shodan search engine[2] is used to find computers running the specific software and version. Shodan lists the banner for the software queried, and these banners are saved in the Python database file. This method is much quicker than examining each Metasploit exploit to see how the version checking is done and manually constructing banners that match these checks. Furthermore, manually constructed banners that merely pass the Metasploit exploit's version check can still be recognized by an attacker who inspects the banner, possibly leading to the attacker not trying to exploit the honeypot.

By default, the script will bind itself to a TCP port when started, and every incoming connection will get a random FTP server banner served. This is chosen so that all FTP server software that contains Metasploit-able vulnerabilities is emulated, which should attract more activity than a single vulnerable FTP server software.

An other option is to have one single FTP banner, containing the names and version numbers of all the vulnerable FTP software. This will allow Metasploit, in theory, to always get a positive match when version checking the banner returned by the FTP honeypot script. However, when an attacker would manually inspect of the banner, it will most likely arouse suspicion. Another problem might be that the resulting banner gets too long, possibly causing an overflow in a connecting client, resulting in a crash of the client. The downsides to this approach are significant enough to choose for random FTP banners being served to the client upon connecting.

## 4.2 Capturing Exploit Traffic

As stated in Section 3.1, using tcpdump to capture all traffic is not a feasible approach. In this research, a different approach is chosen to capture the traffic. The created honeypot script which emulates the FTP service saves all traffic from/to the honeypot itself. Each separate FTP banner has a separate subdirectory in which all traffic coming to/from this FTP banner is saved. Within these subdirectories, all traffic occurring within a single session is saved in a separate subdirectory labeled with the attacker's IP and date/time of connecting. This gives more flexibility in categorizing traffic flows according to time, attacker IP, and allows for each flow in traffic (every request or response to/from the honeypot counts as one single flow) to be analyzed without having to extract a specific flow from a PCAP file.

## 4.3 Detecting Exploit Traffic

Exploit traffic against FTP servers usually consists of multiple request/response flows before the actual exploit is executed. Some of these flows are too generic to include in a signature. An example is the FTP login procedure, which consists of the client sending the command *USER <username>*, followed by the command *PASS <password>*. Since these commands are merely part of getting the FTP connection in a state where the exploit traffic can take place, and these commands occur in almost all FTP traffic, these commands must be excluded from the signature.

As stated in Section 4.1, according to RFC959 all FTP commands and their parameters should only contain 7-bit ASCII characters. When monitoring network traffic during tests, it became clear that every exploit exceeds this character set. Based on this, for each flow that the honeypot FTP script saves, a check is done to see if the flow contains characters outside the 7-bit ASCII character set, and if so, this flow is saved as a suspicious flow, indicating abnormal traffic which in this situation most likely means exploit traffic. Testing this method is done by firing every exploit in the test set two times against the FTP honeypot script, each time with a randomly chosen payload. This testing shows that 100% of the flows containing payloads were flagged as suspicious, as can be seen in Table 4.1.

Two exploits did not complete successfully, one due to the fact that Metasploit could

---

[2] Shodan - Computer Search Engine: http://www.shodanhq.com/

Table 4.1: Number of detected suspicious flows

| Number of exploits executed | Number of suspicious flows detected |
|---|---|
| 36 | 72 |

not find a payload suitable for the exploit, and the second exploit contains a faulty version checking function, expecting a target different than the target the exploit is aimed at. These exploits are not included in the test mentioned above.

## 4.4 Extracting Signatures from Exploit Traffic

Given the different methods for pattern extraction described in Section 3.3, the data has to be analyzed to determine which method suits the input data best.

The data in the network stream consists of long strings, for which numerical features have to be found. Given the fact that running a single exploit with different payloads can result in flows with different lengths, the length of the data is not a good feature to use. Another commonly used feature for string data is the so called edit distance, or the Levenshtein distance. This metric indicates the difference between two strings: the more two strings differ, the higher the edit distance. Since almost all payloads in the flows are encoded using Metasploit's polymorphic encoders resulting in payloads that differ as much as possible from one another, edit distance is very unusable as a feature.

Given the short time span during which this research was done, static pattern analysis was chosen to extract signatures from exploit traffic. This method proved to be the easiest to set up, but relies on static parts occurring in exploit traffic, which is not the case for all exploits in the test set for this report.

The capturing of flows and flagging of suspicious flows as described in Section 4.2 provides a good method to collect input for the LCS algorithm to extract signatures from the suspicious flows per exploit. In order to attempt to obtain a signature, one exploit at a time is executed multiple times against the FTP honeypot script, with a randomly chosen payload each time. Since payloads get encoded every time an exploit runs, this provides for the maximum amount of entropy in the payload section of the exploit traffic, resulting in signatures that should match to the static parts of the exploit traffic only. A custom written Python script is used that locates all suspicious flows inside a specified directory, and runs the LCS algorithm on the suspicious flows. The resulting signature is then printed out and can be imported into the FTP honeypot script signature database. This script can be found in Appendix E.

## 4.5 Matching Exploit Traffic against Signatures

With the signatures generated, the FTP honeypot script has to be adjusted to store and detect the signatures. To do so, the Python database file containing the different banners was adjusted to store the extracted signatures per banner. The FTP honeypot script was adjusted so that every incoming flow is inspected to see if any of the stored signatures occur in the flow. Given that the signatures often contain characters outside the ASCII character set, all signatures are saved in hexadecimal format, and the hexadecimal representation of the incoming flow is then matched against every signature to attempt to find a match. When a match is found, the FTP honeypot script will print out which exploit was detected.

# 5 Findings

## 5.1 Extracting Patterns from Exploit Traffic

Each individual exploit in the test set was put through the pattern extraction method described in Section 4.4. The accuracy of the extracted signature using the LCS algorithm gets higher as more flows from different connections are available as input. This is because the encoded payload data mutates every time the exploit is ran against the FTP honeypot script, causing the LCS algorithm to only return the static parts of the exploit traffic. Because the payload is random every time an exploit is executed, the LCS algorithm already converges to a decent signature as soon as with only two input flows. However, to have more certainty that the resulting signature is indeed good, every exploit was fired 10 times with 10 different payloads against the FTP honeypot script before running the LCS algorithm.

As the length of the extracted pattern gets lower when the number of connections, and thus the number of input flows rises, the chance of false negatives goes down. This is because some exploits contain multiple targets (e.g. the same software installed on different Microsoft Windows versions), which use different hardcoded return addresses in the exploit. With a longer pattern length, one return address might still be present in the extracted pattern, preventing other return addresses from being matched by the extracted pattern. This problem is simply solved by running the exploit numerous times for each target against the FTP honeypot script.

Within the exploit set used in this research project, some exploits can target different versions of the FTP software emulated by the FTP honeypot script. Some of these exploits fail to execute if the selected target version does not match the banner used in the FTP honeypot script.

Not all exploits contain static patterns within the suspicious flow data, or the static pattern is so small that the LCS algorithm returns FTP commands such as "USER" or "PASS" as the longest static pattern within all the suspicious flows. Since these strings are merely part of the FTP protocol, they are not suitable to act as signatures. To attempt to find a solution for this problem, for those exploits which did not contain a suitable signature in the suspicious flow data the LCS algorithm was run on all flows instead of only the suspicious flows. This was done in such a manner that the incoming flows with flow number 1 for all connections were run through the LCS algorithm, followed by running the LCS algorithm on all incoming flows with flow number 2, etc. For each set of flows the outcome of the LCS algorithm was printed, and out of the 17 exploits that did not contain a suitable signature in the suspicious

flows, for 12 exploits a signature was extracted from their remaining flows.

## 5.2 Matching Exploit Traffic against Signatures

To test the extracted signatures, each exploit for which a signature was found was run against the FTP honeypot script using every possible combination of targets and payloads using the Python script in Appendix F. The output from the FTP honeypot script was then analyzed to count the number of connections, and the number of detected exploit attempts. Since some exploits send a payload multiple times per exploit attempt (e.g. in order to brute-force a return address), the results for all exploits were adjusted in such a way that a maximum of one exploit detection per connection was counted.

Out of the 37 exploits in the test set, 3 signatures were extracted that matched more than one exploit. To prevent multiple detection messages, the FTP honeypot script was adjusted so that only exploits for the currently emulated FTP software are matched against the traffic.

Another problem that was faced was that some signatures contained other signatures as a substring. To prevent the FTP honeypot script from returning multiple signatures, different in length, the choice was made to return only the longest matching signature when the traffic matched against multiple signatures. The matching signature with the highest length is also the most specific signature, and it can therefore be assumed that this is the most likely match against the traffic.

To match the accuracy of the extracted signatures, each exploit was fired against the FTP honeypot script, with every possible combination of targets and payloads. The output of the FTP honeypot script was saved per exploit, and the number of connections and the number of matched signatures in the traffic were used to calculate a detection rate. For exploits that have specific FTP software versions set as targets, some targets would cause the exploit to abort right after connecting to the FTP honeypot script, because of a non-matching banner. To account for this, all exploit attempts that aborted after the server presented its banner to the exploit were not included in the results. The full results on detection rates per exploit can be found in Appendix B. A graph showing the detection rates versus the number of exploits that had a signature can be seen in Figure 5.1.

*Figure 5.1: Detection rates versus number of exploits*



15
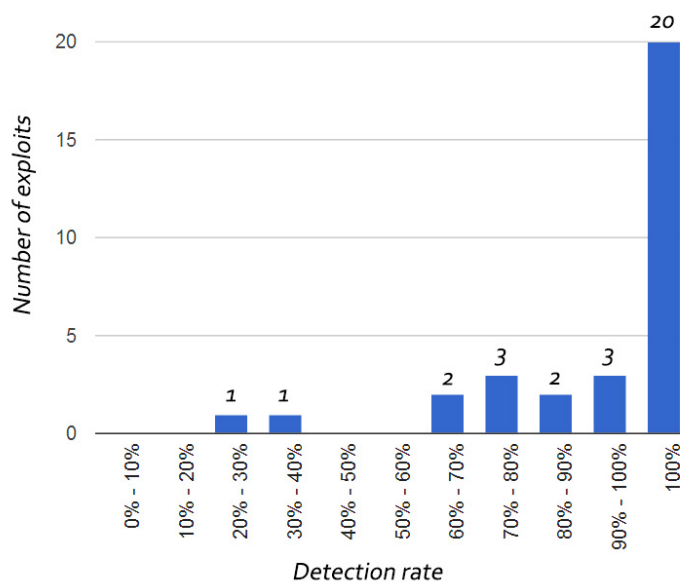
# 6 Conclusions

The research question posed was:

***How feasible is an automated method to detect specific exploits on a honeypot by monitoring network traffic of exploits?***

The sub-questions that followed from this question were:

- What setup is needed in order to have exploits successfully complete their exploit against a honeypot?

- What is the best method to process network traffic to/from the honeypot to extract and match a unique signature from exploit traffic?

- How successful are these methods?

In this research it became clear that the first hurdle of firing exploits against a honeypot is that exploits can be very specific when it comes to the responses they expect from the victim system, both in FTP version banner and the honeypot's response to certain FTP commands. Testing each exploit to be implemented in the honeypot against either a script emulating an FTP server or the actual vulnerable FTP software gives insight into which commands need to be implemented. The FTP version banner can be obtained by looking at existing FTP servers that use the targeted FTP software version.

The stored network data should be as granular as possible, in order to allow analysis on very specific parts of the stored network data. Saving network data per flow, categorized per connection the flows took place in, proved to work well for this research. Detecting actual exploit traffic within flows can be achieved by looking for anomalies within the characters used within the flow. For the FTP protocol used within this paper, the occurrence of non-ASCII characters proved to be a good method to detect exploit traffic. With the stored flows, the Longest Common Substring can be used to obtain signatures from the network flows. This algorithm will only return useful results when static patterns occur within the exploit, which proved the case for 86% of the tested exploits in this research. The signatures can be implemented in the honeypot to match incoming traffic on-the-fly against these signatures, dropping the need for honeypot administrators to perform manual analysis on the stored network traffic in order to detect specific exploits.

The methods described in this paper proved to be very successful. Out of the 37 exploits in the test set used within this paper, 32 exploits yielded a signature after putting the exploits through the signature extraction process. Testing these signatures proved an average detection rate of 89.95%.

To summarize, the method described in this paper seems very feasible, combining high detection rates and a large success rate of obtaining signatures for the tested exploits.

The described method can easily be applied to other protocols.

# 7 Future Work

Due to time constraints, some points could not be researched in this research project.

Porting the FTP honeypot script used within this paper to more protocols would be a valuable addition.

Using other algorithms and/or methods to process the stored network traffic might increase performance compared to the Longest Common Substring algorithm used within this research.

# 8 Acknowledgements

# A Metasploit FTP Exploits Used

| Exploit identifier | Exploit description |
| --- | --- |
| 3cdaemon_ftp_user | 3Com 3CDaemon 2.0 FTP Username Overflow |
| ability_server_stor | Ability Server 2.34 STOR Command Stack Buffer Overflow |
| cesarftp_mkd | Cesar FTP 0.99g MKD Command Buffer Overflow |
| comsnd_ftpd_fmtstr | ComSndFTP v1.3.7 Beta USER Format String (Write4) Vulnerability |
| dreamftp_format | BolinTech Dream FTP Server 1.02 Format String |
| easyfilesharing_pass | Easy File Sharing FTP Server 2.0 PASS Overflow |
| easyftp_cwd_fixret | EasyFTP Server <= 1.7.0.11 CWD Command Stack Buffer Overflow |
| easyftp_list_fixret | EasyFTP Server <= 1.7.0.11 LIST Command Stack Buffer Overflow |
| easyftp_mkd_fixret | EasyFTP Server <= 1.7.0.11 MKD Command Stack Buffer Overflow |
| filecopa_list_overflow | FileCopa FTP Server pre 18 Jul Version |
| freefloatftp_user | Free Float FTP Server USER Command Buffer Overflow |
| freeftpd_user | freeFTPd 1.0 Username Overflow |
| globalscapeftp_input | GlobalSCAPE Secure FTP Server Input Overflow |
| goldenftp_pass_bof | GoldenFTP PASS Stack Buffer Overflow |
| httpdx_tolog_format | HTTPDX tolog() Function Format String Vulnerability |
| ms09_053_ftpd_nlst | Microsoft IIS FTP Server NLST Response Overflow |
| netterm_netftpd_user | NetTerm NetFTPD USER Buffer Overflow |
| oracle9i_xdb_ftp_pass | Oracle 9i XDB FTP PASS Overflow (win32) |
| oracle9i_xdb_ftp_unlock | Oracle 9i XDB FTP UNLOCK Overflow (win32) |
| proftpd_133c_backdoor | ProFTPD-1.3.3c Backdoor Command Execution |
| proftp_sreplace | ProFTPD 1.2 - 1.3.0 sreplace Buffer Overflow (Linux) |
| proftp_telnet_iac | ProFTPD 1.3.2rc3 - 1.3.3b Telnet IAC Buffer Overflow (Linux) |
| ricoh_dl_bof | Ricoh DC DL-10 SR10 FTP USER Command Buffer Overflow |
| sami_ftpd_list | Sami FTP Server LIST Command Buffer Overflow |
| sami_ftpd_user | KarjaSoft Sami FTP Server v2.02 USER Overflow |
| servu_chmod | Serv-U FTP Server < 4.2 Buffer Overflow |
| servu_mdtm | Serv-U FTPD MDTM Overflow |
| slimftpd_list_concat | SlimFTPd LIST Concatenation Overflow |
| turboftp_port | Turbo FTP Server 1.30.823 PORT Overflow |
| vermillion_ftpd_port | Vermillion FTP Daemon PORT Command Memory Corruption |
| vsftpd_234_backdoor | VSFTPD v2.3.4 Backdoor Command Execution |

| Exploit identifier | Exploit description |
| --- | --- |
| warftpd_165_pass | War-FTPD 1.65 Password Overflow |
| warftpd_165_user | War-FTPD 1.65 Username Overflow |
| wftpd_size | Texas Imperial Software WFTPD 3.23 SIZE Overflow |
| wsftp_server_503_mkd | WS-FTP Server 5.03 MKD Overflow |
| wsftp_server_505_xmd5 | Ipswitch WS_FTP Server 5.05 XMD5 Overflow |
| wuftpd_site_exec_format | WU-FTPD SITE EXEC/INDEX Format String Vulnerability |
| xlink_server | Xlink FTP Server Buffer Overflow |

# B Detection Rate of Exploits

| Exploit identifier | Times executed | Times detected | Detection rate | Signature length |
|---|---|---|---|---|
| 3cdaemon_ftp_user | 535 | 518 | 96.8% | 8 bytes |
| ability_server_stor | 214 | 214 | 100% | 4 bytes |
| cesarftp_mkd | 28 | 28 | 100% | 675 bytes |
| comsnd_ftpd_fmtstr | 214 | 214 | 100% | 221 bytes |
| dreamftp_format | 0 | 0 | 0% | 47 bytes |
| easyfilesharing_pass | 216 | 216 | 100% | 4 bytes |
| easyftp_cwd_fixret | 980 | 740 | 75.5% | 36 bytes |
| easyftp_list_fixret | 106 | 89 | 83.9% | 20 bytes |
| easyftp_mkd_fixret | 1059 | 820 | 77.4% | 18 bytes |
| filecopa_list_overflow | 168 | 168 | 100% | 7 bytes |
| freefloatftp_user | 98 | 98 | 100% | 6 bytes |
| freeftpd_user | N/A | N/A | N/A | N/A* |
| globalscapeftp_input | 98 | 96 | 97.9% | 142 bytes |
| goldenftp_pass_bof | 294 | 294 | 100% | 4 bytes |
| httpdx_tolog_format | 749 | 533 | 71.2% | 136 bytes |
| ms09_053_ftpd_nlst | 300 | 300 | 100% | 32 bytes |
| netterm_netftpd_user | 535 | 535 | 100% | 6 bytes |
| oracle9i_xdb_ftp_pass | 108 | 108 | 100% | 7 bytes |
| oracle9i_xdb_ftp_unlock | N/A | N/A | N/A | N/A* |
| proftpd_133c_backdoor | N/A | N/A | N/A | N/A* |
| proftp_sreplace | 63 | 42 | 66.6% | 100 bytes |
| proftp_telnet_iac | 88 | 84 | 95.4% | 3789 bytes |
| ricoh_dl_bof | 107 | 107 | 100% | 4 bytes |
| sami_ftpd_list | 107 | 107 | 100% | 6 bytes |
| sami_ftpd_user | N/A | N/A | N/A | N/A* |
| servu_chmod | 216 | 216 | 100% | 39 bytes |
| servu_mdtm | 321 | 321 | 100% | 81 bytes |
| slimftpd_list_concat | 99 | 99 | 100% | 4 bytes |
| turboftp_port | 301 | 87 | 28.9% | 402 bytes |
| vermillion_ftpd_port | 317 | 209 | 65.9% | 22 bytes |
| vsftpd_234_backdoor | N/A | N/A | N/A | N/A* |
| warftpd_165_pass | 84 | 84 | 100% | 4 bytes |
| warftpd_165_user | 336 | 336 | 100% | 4 bytes |
| wftpd_size | 300 | 300 | 100% | 4 bytes |
| wsftp_server_503_mkd | 99 | 99 | 100% | 8 bytes |
| wsftp_server_505_xmd5 | 81 | 27 | 33.3% | 4 bytes |
| wuftpd_site_exec_format | 84 | 72 | 85.7% | 888 bytes |
| xlink_server | 24 | 24 | 100% | 7 bytes |

* No signature available

# FTP Honeypot Script

```
1    #!/usr/bin/python
2
3    import argparse
4    import copy
5    import math
6    import os
7    import random
8    import re
9    import socket
10   import SocketServer
11   import string
12   import sys
13   import time
14
15   import ftp_db
16
17   # class to handle incoming connections
18   class FTPHandler(SocketServer.BaseRequestHandler):
19
20           def handle(self):
21
22                   client_ip = self.client_address[0]
23
24                   # if user supplied an argument for the FTP banner to be used,
25                   # use this banner. If not, pick a random banner
26                   if args['banner'] and ftp_db.banners[args['banner']]:
27                           ftp_info = ftp_db.banners[args['banner']]
28                           ftp_banner = ftp_info['banner']
29                           ftp_name = args['banner']
30                   else:
31                           ftp_name, ftp_info = random.choice(ftp_db.banners.items())
32                           ftp_banner = ftp_info['banner']
33
34                   # if the dir for this banner does not exist, create it.
35                   if not os.path.isdir(args['flows_dir'] + ftp_name):
36                           os.mkdir(args['flows_dir'] + ftp_name)
37
38                   # set up basic stuff, create flow dir
39                   path = "/"
40                   flow_nr = 0
41                   flow_dir = args['flows_dir'] + ftp_name + "/" + client_ip + "_" + str(time.time())
42                   os.mkdir(flow_dir)
43
44                   # send back initial banner to the client
45                   # print "Connection from " + client_ip + ", serving banner for " + ftp_name
46                   logfile.write("Connection from " + client_ip + ", serving banner for " + ftp_name + "\n")
47                   logfile.flush()
48                   self.send_response(flow_dir, flow_nr, ftp_banner + "\r\n")
49
50                   flow_nr += 1
51
52                   while 1:
53                           found_exploit = ""
54                           found_signature = ""
55
56                           # try to grab all the data from the client in one go,
57                           # so the data does not get split up over multiple flows.
58                           # if there is no data, client aborted the connection.
59                           self.data = self.request.recv(9218).strip()
60                           if not self.data:
61                                   break
62
63                           # due to non-printable characters in exploit traffic, all signatures
64                           # are saved using hexadecimal encoding. we also convert client
65                           # requests to hexadecimal to match for signatures.
66                           # only return the longest matching signature.
67                           for name, signature in ftp_info['signatures'].items():
68                                   if self.data.encode("hex").find(signature) != -1:
69                                           if not found_exploit or len(signature) > ↵
70                                           len(ftp_info['signatures'][found_exploit]):
71                                                   found_exploit = name
72                                                   found_signature = signature
73
74
75                           if found_exploit and found_signature:
76                                   # print "Found exploit " + found_exploit + ": " + found_signature
77                                   logfile.write("Found exploit " + found_exploit + ": " + found_signature + "\n")
78                                   logfile.flush()
79
80                           # All data should only contain 7-bit ASCII data. If this is not
81                           # the case, most likely exploit traffic => mark as suspicious.
82                           if all(ord(c) < 127 and c in string.printable for c in self.data):
83                                   flow = open(flow_dir + "/req_" + str(flow_nr) + ".flow", "w")
84                           else:
85                                   flow = open(flow_dir + "/req_" + str(flow_nr) + ".flow_suspicious", "w")
86                           flow.write(self.data)
87                           flow.close()
88
89                           # extract the FTP command issued by the client, and try to give
90                           # it the response it expects.
91                           cmd = self.data.split(" ")[0].upper()
92
93                           if re.match("USER", cmd):
94                                   user = self.data.split(" ")[-1]
95                                   self.send_response(flow_dir, flow_nr, "331 Password required for " + ↵
96                                   user + "\r\n")
97                           elif re.match("PASS", cmd):
98                                   self.send_response(flow_dir, flow_nr, "230 User " + ↵
99                                   user + " logged in\r\n")
100                          elif re.match("HELP", cmd):
```

```
101                                     self.send_response(flow_dir, flow_nr, "214 Help.\r\n")
102                             elif re.match("QUIT", cmd):
103                                     self.send_response(flow_dir, flow_nr, "221 Goodbye\r\n")
104                                     self.request.close()
105                                     return
106                             elif re.match("SYST", cmd):
107                                     self.send_response(flow_dir, flow_nr, "215 UNIX Type: L8\r\n")
108                             elif re.match("X?MKD", cmd):
109                                     self.send_response(flow_dir, flow_nr, "257 " + self.data.split(" ")[1] + ←↩
110                                     " directory created.\r\n")
111                             elif re.match("PWD", cmd):
112                                     self.send_response(flow_dir, flow_nr, "257 \"" + path + ←↩
113                                     "\" is the current directory.\r\n")
114                             elif re.match("CWD", cmd):
115                                     # allow the client to change dirs, but no escaping the
116                                     # non−existent root dir.
117                                     cwd_dir = self.data.split(" ")[1]
118                                     if cwd_dir == "..":
119                                             if path == "/":
120                                                     path = "/"
121                                             else:
122                                                     (path, tail) = os.path.split(path)
123                                     else:
124                                             path = os.path.join(path, cwd_dir)
125                                     self.send_response(flow_dir, flow_nr, "250 CWD command successful\r\n")
126                             elif re.match("SITE EXEC", self.data.upper()):
127                                     # emulate a SITE EXEC command, simply print back
128                                     # everything after SITE EXEC with a 200 status code.
129                                     exec_cmd = " ".join(self.data.split(" ")[2:])
130                                     self.send_response(flow_dir, flow_nr, "200 " + exec_cmd + "\r\n")
131                             elif re.match("DELE", cmd):
132                                     self.send_response(flow_dir, flow_nr, "250 File deleted.\r\n")
133                             elif re.match("RMD", cmd):
134                                     self.send_response(flow_dir, flow_nr, "250 Directory deleted.\r\n")
135                             elif re.match("TYPE", cmd):
136                                     self.send_response(flow_dir, flow_nr, "200 Type set to " + ←↩
137                                     self.data.split(" ")[1].upper() + "\r\n")
138                             else:
139                                     self.send_response(flow_dir, flow_nr, "502 Command not implemented \r\n")
140
141                             flow_nr += 1
142
143         # send the response back to the client,
144         # save the response in the flow dir
145         def send_response(self, flow_dir, flow_nr, msg):
146                 flow = open(flow_dir + "/resp_" + str(flow_nr) + ".flow", "w")
147                 flow.write(msg)
148                 flow.close()
149
150                 self.request.sendall(msg)
151
152
153 def parseArgs(argv):
154         parser = argparse.ArgumentParser(description='FTP Honeypot script.')
155         parser.add_argument('−−logfile', default='honeypot.log', help="Log file to write messages to.")
156         parser.add_argument('−−port', default='21', help="TCP port number to bind to.")
157         parser.add_argument('−−flows−dir', default='flows/', help="Directory in which to save flows.")
158         parser.add_argument('−−banner', default='', help="FTP banner to serve to clients. If left empty, " +
159                 "random banner will be served. Use −−list−banners to see all banners.")
160         parser.add_argument('−−list−banners', action='store_true', ←↩
161         help="List all banners to use with the −−banner parameter.")
162         args = parser.parse_args(argv)
163         return [vars(args), parser]
164
165
166 def printBanners():
167         print "  Banner                          Description"
168         print "====================================================="
169
170         for banner, banner_info in sorted(ftp_db.banners.items()):
171                 print "  {:<25}{}".format(banner, banner_info['name'])
172
173         print ""
174         print "Use value listed in the 'Banner' column as parameter for the −−banner argument"
175
176
177 if __name__ == "__main__":
178         global parser, args, logfile
179         args, parser = parseArgs(sys.argv[1:])
180
181         if args['list_banners']:
182                 printBanners()
183                 sys.exit(0)
184
185         if args['banner'] and not args['banner'] in ftp_db.banners.keys():
186                 print "Error: this banner does not exist."
187                 print "Specify a correct banner or use the −−list−banners argument to list all valid banners."
188                 sys.exit(−1)
189
190         logfile = open(args['logfile'], "w")
191
192         # Create the server, binding to all interfaces on port specified by args
193         try:
194                 server = SocketServer.TCPServer(("", int(args['port'])), FTPHandler)
195         except socket.error as msg:
196                 print "Error: could not start server (" + msg.strerror + ")"
197                 sys.exit(−1)
198
199         # Activate the server, this will keep running until Ctrl−C'd
200         server.serve_forever()
```

# D    FTP Honeypot Database

```python
 1  banners = {
 2      "3cdaemon": {
 3          "name": "3Com 3CDaemon FTP Server 2.0",
 4          "banner": "220 3Com 3CDaemon FTP Server Version 2.0",
 5          "signatures": {
 6              "exploit/windows/ftp/3cdaemon_ftp_user (Windows 2000 English)": "c42a0275",
 7              "exploit/windows/ftp/3cdaemon_ftp_user (Windows XP English SP0/SP1)": ↩
 8              "d9eed97424f45b817313",
 9              "exploit/windows/ftp/3cdaemon_ftp_user (Windows NT 4.0 SP4/SP5/SP6)": "99176877",
10              "exploit/windows/ftp/3cdaemon_ftp_user (Windows 2000 Pro SP4 French)": "d0295f77",
11              "exploit/windows/ftp/3cdaemon_ftp_user (Windows XP English SP3)": "fb41bd7cfb41bd7c"
12          }
13      },
14      "ability": {
15          "name": "Ability FTP Server 2.34",
16          "banner": ↩
17          "220 Welcome to Code−Crafters − Ability Server 2.34. (Ability Server 2.34 by Code−Crafters).",
18          "signatures": {
19              "exploit/windows/ftp/ability_server_stor (Windows XP SP2 ENG)": "cf2ee373",
20              "exploit/windows/ftp/ability_server_stor (Windows XP SP3 ENG)": "5393427e"
21          }
22      },
23      "cesarftp": {
24          "name": "Cesar FTP Server 0.99g",
25          "banner": "220 CesarFTP 0.99g Server Welcome !",
26          "signatures": {
27              "exploit/windows/ftp/cesarftp_mkd": "4d4b4420" + "0a" * 671
28          }
29      },
30      "comsnd": {
31          "name": "ComSnd FTP Server 1.3.7",
32          "banner": "220 " + "32323020bbb6d3adb9e2c1d9465450b7fecef1c6f7210d0a"↩
33          .decode("hex"), # ugly non ASCII chars in banner
34          "signatures": {
35              "exploit/windows/ftp/comsnd_ftpd_fmtstr (Windows XP SP3 − English)":↩
36              "5040ac71253432333034323278" + "2570" * 208,
37              "exploit/windows/ftp/comsnd_ftpd_fmtstr (Windows Server 2003 − English)":↩
38              "4440c171253432333034323278" + "2570" * 208,
39          }
40      },
41      "dreamftp": {
42          "name": "BolinTech Dream FTP Server 1.02",
43          "banner": "220 DreamHost FTP Server",
44          "signatures": {
45              "exploit/windows/ftp/dreamftp_format": "eb29253878253878253878253878253878825388" + ↩
46              "78253782538782533393535736383064256e256e4040404040404040"
47          }
48      },
49      "easyfilesharing": {
50          "name": "Easy File Sharing FTP Server 2.0",
51          "banner": "220 Welcome to Easy File Sharing FTP Server!",
52          "signatures": {
53              "exploit/windows/ftp/easyfilesharing_pass (Windows 2000 Pro English ALL)": "c42a0275",
54              "exploit/windows/ftp/easyfilesharing_pass (Windows XP Pro SP0/SP1 English)": "ad32aa71"
55          }
56      },
57      "easyftp": {
58          "name": "BigFoolCat FTP Server 1.0",
59          "banner": "220− Ftp Site Powerd by BigFoolCat Ftp Server 1.0 (meishu1981@gmail.com)\r\n" +
60          "220− Welcome to my ftp server\r\n" +
61          "220",
62          "signatures": {
63              "exploit/windows/ftp/easyftp_cwd_fixret": "89e781ef14feffff890f81c714ffffffffe7",
64              "exploit/windows/ftp/easyftp_list_fixret": "ffff81c404feffffffe7",
65              "exploit/windows/ftp/easyftp_mkd_fixret": "81c714ffffffffe743434343434343434343"
66          }
67      },
68      "filecopa": {
69          "name": "FileCOPA FTP Server 1.01",
70          "banner": "220−InterVations FileCOPA FTP Server Version 1.01 21st November 2005",
71          "signatures": {
72              "exploit/windows/ftp/filecopa_list_overflow": "6681c1a00151c3"
73          }
74      },
75      "freefloatftp": {
76          "name": "FreeFloat FTP Server 1.00",
77          "banner": "220 FreeFloat Ftp Server (Version 1.00).",
78          "signatures": {
79              "exploit/windows/ftp/freefloatftp_user": "81c454f2ffff"
80          }
81      },
82      "globalscapeftp": {
83          "name": "GlobalSCAPE Secure FTP Server 3.2",
84          "banner": "220 GlobalSCAPE Secure FTP Server (v. 3.2)",
85          "signatures": {
86              "exploit/windows/ftp/globalscapeftp_input": "eb0359eb05e8f8ffffff565458363" + ↩
87              "3305754583633385658434948484850565835414151515056583559595959595035595959" + ↩
88              "44354b4b594150545458363338544444e56444458345a344136333836313831364949494" + ↩
89              "94949494949494949515a565458333056583441503041334848304130304142414142544141" + ↩
90              "5132414232424230424258503841434a4a49"
91          }
92      },
93      "goldenftp": {
94          "name": "Golden FTP Server 4.70",
95          "banner": "220 Golden FTP Server ready v4.70",
96          "signatures": {
97              "exploit/windows/ftp/goldenftp_pass_bof": "d97424f4"
98          }
99      },
100     "httpdx": {
```

```
101              "name": "Httpdx FTP Server 1.4",
102              "banner": "220 httpdx/1.4 (Win32)",
103              "signatures": {
104                      "exploit/windows/ftp/httpdx_tolog_format": "e870f864" + "253878" * 37  + ←
105                      "25303235353227825686e2530343838387825686e",
106              }
107      },
108      "ms09": {
109              "name": "Microsoft IIS FTP server 5.0",
110              "banner": "220 localhost Microsoft FTP Service (Version 5.0).",
111              "signatures": {
112                      "exploit/windows/ftp/ms09_053_ftpd_nlst (Windows 2000 SP4 English/Italian (IIS 5.0))": ←
113                      "fd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7feb243d77fd7ffd7ffd7ffd7ffd7f",
114                      "exploit/windows/ftp/ms09_053_ftpd_nlst (Windows 2000 SP3 English (IIS 5.0))": ←
115                      "fd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7fd82ee477fd7ffd7ffd7ffd7ffd7f",
116                      "exploit/windows/ftp/ms09_053_ftpd_nlst (Windows 2000 SP0−SP3 Japanese (IIS 5.0))": ←
117                      "fd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7ffd7f93a54f77fd7ffd7ffd7ffd7ffd7f"
118              }
119      },
120      "netterm": {
121              "name": "Netterm FTP Server",
122              "banner": "220 NetTerm FTP server ready",
123              "signatures": {
124                      "exploit/windows/ftp/netterm_netftpd_user": "5553455220c0"
125              }
126      },
127      "oracle9i": {
128              "name": "Oracle 9i XML Database FTP Server 9.2.0.1",
129              "banner": "220 localhost FTP Server (Oracle XML DB/Oracle9i Enterprise Edition " + ←
130              "Release 9.2.0.1.0 − Production) ready",
131              "signatures": {
132                      "exploit/windows/ftp/oracle9i_xdb_ftp_pass": "81c4ffefffff44"
133              }
134      },
135      "proftpd_130": {
136              "name": "ProFTPD FTP Server 1.3.0",
137              "banner": "220 ProFTPD 1.3.0 Server (ProFTPD Default Installation) [127.0.0.1]",
138              "signatures": {
139                      "exploit/linux/ftp/proftp_sreplace": "42" * 64 + "c8af0408f4590b08" + "cc" * 28
140              }
141      },
142      "proftp": {
143              "name": "ProFTPD FTP Server 1.3.3a",
144              "banner": "220 ProFTPD 1.3.3a Server (ProFTPD Default Installation) [127.0.0.1]",
145              "signatures": {
146                      "exploit/linux/ftp/proftp_telnet_iac": "ff" * 3789
147              }
148      },
149      "ricoh": {
150              "name": "DSC FTP Server 1.0",
151              "banner": "220 DSC ftpd 1.0 FTP Server [127.0.0.1] ready",
152              "signatures": {
153                      "exploit/windows/ftp/ricoh_dl_bof": "5954c377"
154              }
155      },
156      "sami": {
157              "name": "Sami FTP Server 2.0.2",
158              "banner": "220−Sami FTP Server 2.0.2\r\n" +
159                      "220 Features p a .",
160              "signatures": {
161                      "exploit/windows/ftp/sami_ftpd_list": "81c454f2ffff"
162              }
163      },
164      "servu": {
165              "name": "Serv−U FTP Server 4.1.0.3",
166              "banner": "220−Serv−U FTP Server 4.1.0.3",
167              "signatures": {
168                      "exploit/windows/ftp/servu_chmod": ←
169                      "6681caff0f42526a0258cd2e3c055a74efb85730305489d7af75eaaf75e75131c931c002040f41",
170                      "exploit/windows/ftp/servu_mdtm (Serv−U Uber−Leet Universal ServUDaemon.exe)": ←
171                      "4d44544d20323030333313131313131313131312b41414141414141414141414141414141" + ←
172                      "414183c4fc5fbe333332314647393775fb464f3977fc75faffe74242ebe4424277184000202f35333231",
173                      "exploit/windows/ftp/servu_mdtm (Serv−U 4.0.0.4/4.1.0.0/4.1.0.3 ServUDaemon.exe)": ←
174                      "4d44544d20323030333313131313131313131312b41414141414141414141414141414141" + ←
175                      "414183c4fc5fbe333332314647393775fb464f3977fc75faffe74242ebe442424d164000202f35333231",
176                      "exploit/windows/ftp/servu_mdtm (Serv−U 5.0.0.0 ServUDaemon.exe)": ←
177                      "4d44544d20323030333313131313131313131312b41414141414141414141414141414141" + ←
178                      "414183c4fc5fbe333332314647393775fb464f3977fc75faffe74242ebe442427e164000202f35333231"
179              }
180      },
181      "slimftpd": {
182              "name": "SlimFTPd FTP Server 3.16",
183              "banner": "220−SlimFTPd 3.16, by WhitSoft Development (www.whitsoftdev.com)\r\n" +
184                      "220−You are connecting from 127.0.0.1:12345\r\n" +
185                      "220 Proceed with login.",
186              "signatures": {
187                      "exploit/windows/ftp/slimftpd_list_concat": "d97424f4"
188              }
189      },
190      "turboftp": {
191              "name": "TurboFTP FTP Server 1.30.823",
192              "banner": "220 TurboFTP Server 1.30.823 ready.",
193              "signatures": {
194                      "exploit/windows/ftp/turboftp_port": "2c3133332c32352c36352c302c34322c31303" + ←
195                      "52c37352c302c3131362c39362c39352c302c34322c3234382c37302" + ←
196                      "c302c3134392c35392c36362c302c33392c35382c36362c302c31353" + ←
197                      "32c32382c39332c302c39332c3137332c37362c302c3130372c31373" + ←
198                      "72c37342c302c312c302c302c302c3234362c3234372c39342c302c3" + ←
199                      "02c31362c302c302c35372c3131332c39332c302c36342c302c302c3" + ←
200                      "02c3232342c3234312c37372c302c3133332c32352c36352c302c353" + ←
201                      "72c33382c38302c302c3134342c3134342c3134342c3134342c31353" + ←
202                      "22c3132392c37302c302c3133372c3139342c3132392c3233342c302" + ←
203                      "c31362c302c302c3132392c3233362c302c31362c302c302c3130322" + ←
204                      "c3132392c3230322c3235352c31352c36362c38322c3130362c322c3" + ←
205                      "8382c3230352c34362c36302c352c39302c3131362c3233392c31383" + ←
206                      "42c3131392c34382c34382c3131362c3133372c3231352c3137352c3" + ←
207                      "131372c3233342c3137352c3131372c3233312c38312c34392c32303" + ←
208                      "12c34392c3139322c322c342c31352c36352c31"
209              }
210      },
211      "vermillion": {
212              "name": "Vftpd FTP Server 1.31",
213              "banner": "220 localhost FTP Server (vftpd 1.31) ready.",
214              "signatures": {
215                      "exploit/windows/ftp/vermillion_ftpd_port":←
216                      "2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c2c3137312c3438"
217              }
218      },
219      "warftpd": {
```

```
220            "name": "freeFTPd FTP Server 1.0",
221            "banner": "220 ——freeFTPd 1.0———warFTPd 1.65———",
222            "signatures": {
223                "exploit/windows/ftp/warftpd_165_user (Windows 2000 SP0—SP4 English)": "e2310275",
224                "exploit/windows/ftp/warftpd_165_user (Windows XP SP0—SP1 English)": "541dab71",
225                "exploit/windows/ftp/warftpd_165_user (Windows XP SP2 English)": "7293ab71",
226                "exploit/windows/ftp/warftpd_165_user (Windows XP SP3 English)": "532bab71",
227                "exploit/windows/ftp/warftpd_165_pass": "2b774e5f"
228            }
229        },
230        "wftpd": {
231            "name": "WFTPD Pro FTP Server 3.23",
232            "banner": "220 ProFTPD 1.3.1rc2 Server (WFTPD Pro Server 3.23) [127.0.0.1]",
233            "signatures": {
234                "exploit/windows/ftp/wftpd_size": "d97424f4"
235            }
236        },
237        "wsftp": {
238            "name": "WS_FTP FTP Server 5.0.3",
239            "banner": "220 localhost.localdomain X2 WS_FTP Server 5.0.3 (4278729194)",
240            "signatures": {
241                "exploit/windows/ftp/wsftp_server_503_mkd": "b85b1825b85b1825"
242            }
243        },
244        "wsftp_505": {
245            "name": "WS_FTP FTP Server 5.0.3",
246            "banner": "220 localhost.localdomain X2 WS_FTP Server 5.0.5 (543219441)",
247            "signatures": {
248                "exploit/windows/ftp/wsftp_server_505_xmd5": "63c62e7c"
249            }
250        },
251        "wuftpd": {
252            "name": "WUftpd FTP Server 2.6.0(1)",
253            "banner": ↩
254            "220 localhost.localdomain FTP server (Version wu—2.6.0(1) Mon Feb 28 10:30:36 EST 2000) ready.",
255            "signatures": {
256                "exploit/multi/ftp/wuftpd_site_exec_format": "e7060" + "825387" * 276 + "82530"
257            }
258        },
259        "xlink": {
260            "name": "Omni—NFS x—link FTP Server 5.2",
261            "banner": "220—Microsoft FTP Service\r\n220 x—link ftp server",
262            "signatures": {
263                "exploit/windows/ftp/xlink_server": "81c4ffefffff44"
264            }
265        }
266 }
```

# E Longest Common Substring Script

```python
#!/usr/bin/python

import os
import re
import string
import sys

# from http://stackoverflow.com/a/2894073/1546714
def lcs(data):
        substr = ''
        if len(data) > 1 and len(data[0]) > 0:
                # for every possible substring in data[0], check if this substring
                # also occurs in all other items in the data list. If so, pick the
                # longest string and return it.
                for i in range(len(data[0])):
                        for j in range(len(data[0])-i+1):
                                if j > len(substr) and all(data[0][i:i+j] in x for x in data):
                                        substr = data[0][i:i+j]
        return substr


if len(sys.argv) < 2:
        print "Usage: " + sys.argv[0] + " <directory containing flow dirs> [number of connections to process]"
        sys.exit(-1)


parent_dir = sys.argv[1]
flow_files = {}

# for each flow_dir (connection), append the incoming flow files to the list
# of files to be LCS'd.
for flow_dir in os.listdir(parent_dir):
        for flow_file in os.listdir(parent_dir + "/" + flow_dir):
                match = re.search(r"req_(?P<flownr>[\d]+)\.flow", flow_file)
                if match:
                        flow_nr = int(match.group("flownr"))
                        if not flow_nr in flow_files:
                                flow_files[flow_nr] = []

                        flow_files[flow_nr].append(parent_dir + "/" + flow_dir + "/" + flow_file)

# for each flow in the LCS todo list, read the contents
flow_contents = {}
for flow_nr, flows in flow_files.items():
        flow_contents[flow_nr] = []
        for f in flows:
                flow_data = open(f, "r")
                # strip any FTP commands from the beginning of the flow
                flow_data_stripped = re.sub("^[A-Z5]{3,4}\s", "", flow_data.read().strip())
                flow_contents[flow_nr].append(flow_data_stripped)
                flow_data.close()

# for all the flow data, run LCS per flow number, print out the results
for flow_nr, flows in flow_contents.items():
        if len(flows):
                print "Finding LCS signature for " + str(len(flows)) + " flows on request nr " + str(flow_nr)
                signature = lcs(flows)
                if all(ord(c) < 127 and c in string.printable for c in signature):
                        print "Hexadecimal signature: " + signature.encode("hex") + ", ASCII: " + signature
                else:
                        print "Hexadecimal signature: " + signature.encode("hex")
```

# F Signature Testing Script

```python
1   #!/usr/bin/python
2
3   import ftp_db
4   import os
5   from random import randint
6   import re
7   import signal
8   import subprocess
9   import time
10
11  # get all payloads from msfcli that can be used with this exploit
12  def get_payloads(exploit):
13      output = subprocess.check_output("msfcli " + exploit + " P | grep -E 'windows|linux'" + \
14      " | awk '{print $1}'", shell=True)
15      return [s.strip() for s in output.splitlines()]
16
17  # get all targets from msfcli that can be used with this exploit
18  def get_targets(exploit):
19      targets = []
20      output = subprocess.check_output("msfcli " + exploit + " T 2>&1 | grep -E '[0-9]'", shell=True)
21      for s in output.splitlines():
22          target_match = re.match(r"([\s\t]+)([0-9])(.*)", s)
23          if target_match:
24              targets.append(target_match.group(2))
25      return targets
26
27  # spawn the FTP honeypot script
28  def start_honeypot(banner, flow_dir):
29      global honeypot_proc
30      cur_path = os.path.dirname(os.path.abspath(__file__)) + "/"
31      # ugly, but we NEED it to spawn in the background
32      honeypot_proc = os.system(cur_path + "ftp.py --banner " + banner + " --flows-dir " + flow_dir + " &")
33      print honeypot_proc
34
35  # ugly hack to find the pid of the FTP honeypot script, and then kill it.
36  def stop_honeypot():
37      p = subprocess.Popen(['ps', '-A'], stdout=subprocess.PIPE)
38      out, err = p.communicate()
39
40      for line in out.splitlines():
41          if 'ftp.py' in line:
42              pid = int(line.split(None, 1)[0])
43              os.kill(pid, signal.SIGKILL)
44              print "Killed honeypot with PID " + str(pid)
45              break
46
47  # write the batch files for metasploit used to bulk test
48  def write_metasploit_rc(exploit):
49      exploit_shortname = exploit.split("/")[-1]
50      exploit_rc_files = []
51      payloads = get_payloads(exploit)
52      targets = get_targets(exploit)
53      # one rc file per target, in case we need to extract signatures for each target
54      for target in targets:
55          output_file = open(exploit_shortname + target + ".rc", "w")
56          output_file.write("use " + exploit + "\n")
57          # fill in all possible options for a payload
58          for payload in payloads:
59              rand_port = randint(10000, 30000)
60              output_file.write("set payload " + payload + "\n")
61              output_file.write("set pexec /opt/metasploit/apps/pro/msf3/data/meterpreter/metsvc.exe\n")
62              output_file.write("set cmd cmd\n")
63              output_file.write("set dnszone google.com\n")
64              output_file.write("set dns google.com\n")
65              output_file.write("set dll /opt/metasploit/apps/pro/msf3/data/meterpreter/metsrv.dll\n")
66              output_file.write("set rc4password testtest\n")
67              output_file.write("set lhost localhost\n")
68              output_file.write("set rhost localhost\n")
69              output_file.write("set rport 21\n")
70              output_file.write("set lport " + str(rand_port) + "\n")
71              output_file.write("set target " + str(target) + "\n")
72              output_file.write("run\n")
73          output_file.write("quit\n")
74          output_file.close()
75          exploit_rc_files.append(exploit_shortname + target + ".rc")
76      return exploit_rc_files
77
78  # aaaaaand lets fire the batch file up with msfconsole
79  def run_metasploit_rc(rc_file):
80      p = subprocess.call(['msfconsole', '-r', rc_file])
81
82  # start off with an empty exploit list
83  exploits = []
84
85  # for each banner that we have exploits for, grab all the exploits
86  for banner, ftp_info in ftp_db.banners.items():
87      # for each exploit, create all rc files and run them against the honeypot
88      for exploit in ftp_info['signatures'].keys():
89          if " " in exploit:
90              exploit = exploit.split(" ")[0]
91          if not exploit in exploits:
92              start_honeypot(banner, "testing_flows/")
93              exploits.append(exploit)
94              rc_files = write_metasploit_rc(exploit)
95              for rc_file in rc_files:
96                  run_metasploit_rc(rc_file)
97              stop_honeypot()
98              # save the generated logfile as <exploitname>.log
99              exploit_shortname = exploit.split("/")[-1]
100             os.rename(os.getcwd() + "/honeypot.log", os.getcwd() + "/" + exploit_shortname + ".log")
```

# Bibliography

[1] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, p. 365, 1996.

[2] G. Richarte, "Riq," *Advances in format string exploitation. Phrack Magazine*, vol. 59, no. 7, 2002.

[3] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *Security & Privacy, IEEE*, vol. 2, no. 4, pp. 20–27, 2004.

[4] C. Anley, J. Heasman, F. Lindner, and G. Richarte, *The shellcoder's handbook: discovering and exploiting security holes*. Wiley, 2011.

[5] S. Chong, "History and advances in windows shellcode," *Phrack.[Online]*, vol. 22, 2004.

[6] I. Arce, "The shellcode generation," *Security & Privacy, IEEE*, vol. 2, no. 5, pp. 72–76, 2004.

[7] M. Roesch *et al.*, "Snort-lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration*, pp. 229–238, Seattle, Washington, 1999.

[8] S. Andersson, A. Clark, and G. Mohay, "Network based buffer overflow detection by exploit code analysis," 2004.

[9] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 298–307, ACM, 2004.

[10] M. Prandini and M. Ramilli, "Return-oriented programming," *Security & Privacy, IEEE*, vol. 10, no. 6, pp. 84–87, 2012.

[11] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "Polymorphic shellcode engine using spectrum analysis," 2003.

[12] M. Van Gundy, D. Balzarotti, and G. Vigna, "Catch me, if you can: Evading network signatures with web-based polymorphic worms," in *Proceedings of the First USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[13] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network–level polymorphic shellcode detection using emulation," in *Detection of Intrusions and Malware & Vulnerability Assessment*, pp. 54–73, Springer, 2006.

[14] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. Li, J. Kuo, and K.-P. Fan, "Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities," in *Network Operations and Management Symposium, 2004. NOMS 2004.*

*IEEE/IFIP*, vol. 1, pp. 235–248, IEEE, 2004.

[15] L. Spitzner, *Honeypots: tracking hackers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] N. Provos, "Honeyd-a virtual honeypot daemon," in *10th DFN-CERT Workshop, Hamburg, Germany*, vol. 2, 2003.

[17] SURFcert, "Surfcert ids." `http://ids.surfnet.nl/`. Accessed June 2013.

[18] Dionaea, "Dionaea, catches bugs." `http://dionaea.carnivore.it/`. Accessed June 2013.

[19] Kippo, "Ssh honeypot." `https://code.google.com/p/kippo/`. Accessed June 2013.

[20] J. van der Lelie and R. Breuk, "Visualizing attacks on honeypots," 2012.

[21] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: an automated script generation tool for honeyd," in *Computer Security Applications Conference, 21st Annual*, pp. 12–pp, IEEE, 2005.

[22] C. Kreibich and J. Crowcroft, "Honeycomb: creating intrusion detection signatures using honeypots," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 51–56, 2004.

[23] C. Anley, J. Heasman, F. Lindner, and G. Richarte, *The shellcoder's handbook: discovering and exploiting security holes*. Wiley, 2011.

[24] R. Baumann and C. Plattner, "White paper: Honeypots," 2002.

[25] J. Postel and J. Reynolds, "File transfer protocol," 1985.

[26] L. Rist, S. Vetsch, M. Kossin, and M. Mauer, "Know your tools: Glastopf-a dynamic, low-interaction web application honeypot," *The Honeynet Project*, 2010.

[27] M. Andreolini, A. Bulgarelli, M. Colajanni, and F. Mazzoni, "Honeyspam: Honeypots fighting spam at the source," in *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, pp. 11–11, USENIX Association, 2005.

[28] E. Ramirez-Silva and M. Dacier, "Empirical study of the impact of metasploit-related attacks in 4 years of attack traces," in *Advances in Computer Science–ASIAN 2007. Computer and Network Security*, pp. 198–211, Springer, 2007.

[29] A. K. S. Jethoe, "Snorting metasploit update," 2011.

[30] D. Groenewegen, M. Kuczynski, and J. van Beek, "Offensive technologies," 2011.

[31] C. Jordan, J. ROYES, and J. WHYTE, "Writing detection signatures," *USENIX; login*, vol. 30, no. 6, pp. 55–61, 2005.

[32] G. Vigna, W. Robertson, and D. Balzarotti, "Testing network-based intrusion detection signatures using mutant exploits," in *Proceedings of the 11th ACM conference on Computer and communications security*, pp. 21–30, ACM, 2004.

[33] J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Security and Privacy, 2005 IEEE Symposium on*, pp. 226–241, IEEE, 2005.

[34] M. A. Beddoe, "Network protocol analysis using bioinformatics algorithms," 2005.