Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations

Graduate Studies

5-2014

Automated Reverse Engineering of Malware to Develop Network Signatures to Match with Known Network Signatures

Dan Sinema
Utah State University

Recommended Citation

Sinema, Dan, "Automated Reverse Engineering of Malware to Develop Network Signatures to Match with Known Network Signatures" (2014). *All Graduate Theses and Dissertations*. 3315.
https://digitalcommons.usu.edu/etd/3315

AUTOMATED REVERSE ENGINEERING OF MALWARE TO DEVELOP NETWORK SIGNATURES TO MATCH WITH KNOWN NETWORK SIGNATURES

by

Dan Sinema

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Dan Watson Major Professor	Dr. Ming Li Committee Member
Dr. Nicholas Flann Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

Automated Reverse Engineering of Malware to Develop Network Signatures to Match With Known Network Signatures

by

Dan Sinema, Master of Science

Utah State University, 2014

Major Professor: Dr. Dan Watson
Department: Computer Science

The detection of network-based malware is often reactionary; discovery generally happens after the malware has begun attacking the target system. Detecting the attack after the fact affects the performance of the victim device and potentially the entire computer network of the victim device. Intrusion detection systems are deployed to monitor network traffic for malware attacks, but unfortunately these systems cannot preemptively detect malicious behavior on a network. Automated reverse engineering is able to detect potentially malicious network behavior of a binary offline prior to a network-based attack. Collecting information found inside a binary, such as strings and function calls, compiling this information into generated signatures, and then comparing to known network signatures allows for malicious behavior of a binary to be discovered and quarantined before attacking a device and network.

(72 pages)

PUBLIC ABSTRACT

Automated Reverse Engineering of Malware to Develop Network Signatures to Match With Known Network Signatures

Dan Sinema

Illicit software that seeks to steal user information, deny service, or cause general mayhem on computer networks is often discovered after the damage has been done. The ability to discover network behavior of software before a computer network is utilized would allow administrators to protect and preserve valuable resources. Static reverse engineering is the process of discovering in a offline environment how a software application is built and how it will behave. By automating static reverse engineering, software behavior can be discovered before it is executed on client devices. Fingerprints are then built from the discovered behavior which is matched with known malicious fingerprints to identify potentially dangerous software.

ACKNOWLEDGMENTS

I would like to thank Dr. Chad Mano for teaching class in my first semester in the program that peaked my interest in computer security. I would also like to thank my many committee members for their patience in waiting for me to finish this thesis. And especially Dr. Dan Watson for helping me close this thesis out and pushing me to finish.

I am especially thankful for my family for the encouragement, support, and patience that they showed me over the many late nights and weekends I spent working on this thesis.

Dan Sinema

CONTENTS

Page

ACRONYMS

BSD	Berkeley Software Distribution
C & C	Command & Control
CIDR	Classless Inter-Domain Routing
DNS	Domain Name System
DDoS	Distributed Denial of Service
FLIRT	Fast Library Identification and Recognition Technology
FTP	File Transfer Protocol
HIDS	Host-based Intrusion Detection System
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IDS	Intrusion Detection System
IP	Internet Protocol
IPX	Internetwork Packet Exchange
IRC	Internet Relay Chat
NIDS	Network-based Intrusion Detection System
OSI	Open Systems Interconnections
PE	Portable Executable
TCP	Transmission Control Protocol
UPX	Ultimate Packer for eXecutables

CHAPTER 1

INTRODUCTION

Modern day malware is not isolated to pre-defined exploits like malware of the past [1]. Much of today's malware connects over computer networks to centralized servers known as a command and control servers (C & C servers). These servers send commands to the malware instructing it to execute network or computer based exploits, massive coordinated network attacks, or email spam [2]. When this type of malware infects a computer, the computer becomes known as a zombie, and a collection of zombies is known as a botnet. The communication channel between the zombie and the C & C server is typically standard protocols such as internet relay chat (IRC) or hypertext transfer protocol (HTTP). The botnet is controlled by a botmaster that issues singular commands to run in coordination on the botnet. The command might be a distributed denial of service attack (DDoS), this attack sends millions, upon millions of network packets from the zombies to a targeted device; this deluge of network traffic overwhelms the targeted network device and leads to it becoming unresponsive or possibly crashing.

Criminals and criminal organizations have used the threat of DDoS attack by botnets to extract payments from corporations and organizations [3], recently botnets have been used for political activism, attacking corporate competition, and even nations "weaponizing" botnets to attack a rival nation's infrastructure [4–6]. In a recent case, it is suspected a company in the mining industry and a company in the wine-making industry used DDoS to attack competitor's websites [7]. In addition botnets are now being rented openly on the Internet for relatively insignificant amounts of money [8]. Because of the seriousness of the threat the research of detecting and neutralizing botnets is very relevant and important, and reducing the number of bots will reduce the effectiveness of a botnet.

This thesis will show that an automated approach to reverse engineering of binaries through static analysis is an effective method to build signatures of a binary's network behavior. Embedded in a binary is information that infers how and what it will send on a network; although not as detailed as raw source code, enough information is disclosed to identify network behavior. This information includes system calls, embedded strings, and data flow paths. By using reverse engineering techniques the data needed for network signatures can be identified, traced and extracted without the need to execute a binary and capture live network data. The data when derived and combined together provides enough information to build identifying signatures of a static binary's network behavior. The ability to discover network behavior before the malware is active on a live network can prevent propagation of malware on other computers, denial of service or other exploits, and generally can protect sensitive information. The automation of deriving network signatures from a static binary is the primary contribution of this thesis.

The early chapters (2-4) give background on reverse engineering and the signature format for this thesis. The remaining chapters discuss the reverse engineering process, the data being sought after, and how the data is used to build signatures. Chapter 2 discusses binary obfuscation as related to reverse engineering. Chapter 3 provides an overview of the different reverse engineering methods used today and the tools used to automate the reverse engineering process for this paper. Chapter 4 examines the various network system calls and data contained in a binary that are used to build signatures. Chapter 5 looks at the open source Snort Intrusion Detection System, and Snort rules which will be the format of signatures produced by the implementation of this thesis. Chapter 6 will describe the design and implementation of the network signatures, how the data is gathered and compiled based upon the information described in previous chapters, and finally the process of matching the generated network signatures and the known network signatures. Chapter 6 also includes the results the experiment based on this paper. Chapter 7 presents future work and directions for this topic of research.

1.1 Related Works

There is considerable previous work on the subject of automated signature generation.

Kaur and Singh [9] provide an overview of the current automated signature generation systems, the authors divide the systems into two categories signature generation with attack detection and signature generation without attack detection. The first category is systems such as Honeycyber [10], ARBOR [11], Argos [12], and Hamsa [13] which mainly use dynamic analysis like honetnets and packet tracing for signature generation. The projects that fall in the category of signature generation without attack detection are Hancock [14], F-Sign [15], and Auto-Sign [16]. Hancock utilizes static disassembly to generate string based signatures, the signatures are in a format for antivirus software. F-Sign generates byte-string signatures based upon unique functions contained in the binary, the signatures are in the eDare format. Auto-Sign generates signatures similar to F-Sign but in addition the final signature is based on entropy. All of these systems simply create signatures to be used with external IDSs in this paper not only are signatures auto-generated but those signatures are then compared to known signatures to find matches or highly related signatures. While the other systems rely on IDSs collecting data from a network connection, this paper performs the matching of signatures offline.

CHAPTER 2
BINARY OBFUSCATION

Binary obfuscation is a common technique used by malware developers to frustrate reverse engineers and attempt to prevent detection by anti-malware software. Packers and obfuscators are software applications used to obfuscate malware binaries. In many cases packers and obfuscators are legitimate software products used by legitimate developers to reduce binary file size or protect their code from competitors or software crackers. For example, a developer might use a packer to reduce the size of their application binary for Internet downloading or to frustrate a reverse engineer the developer may use a packer to encrypt parts of an application binary [17]. Obfuscators are primarily used for languages such as Java and .Net, as these languages compile binaries into intermediate byte code to run in virtual machines, and this intermediate byte code can be reversed to Java or .Net source code without much effort. To protect intellectual property from software pirates and competitors, developers often employ obfuscators to "scramble" the byte code making it difficult to reverse the byte code into source code [17].

There are a number of different tools and techniques that can be used to detect if a binary has been packed or obfuscated. A technique used to discover if a PE file (a Windows binary) has been "packed" is to examine the PE file header and look for non-standard PE sections. The UPX packer [18] is an example, that creates non-standard sections UPX0 and UPX1 that are generally not found in a non-packed binary. Figure 2.1 is the Microsoft DUMPBIN utility output of a UPX packed binary and Figure 2.2 is the DUMPBIN output of a non-UPX packed binary for comparison. Another method is to simply try to unpack binaries with the various packing utilities to see if the file will unpack. And finally utilities such as PEfile [19] examine a PE file and compare against known packer signatures to determine if the file is packed and will try to determine which packer packed the file. Seitz demonstrates how to use PyEmu [20] a Python library to extract the UPX0 and UPX1 sections of a UPX packed PE file without the use of the UPX packer [21].

```
C:\upx391w>dumpbin calc_upx.exe
Microsoft (R) COFF/PE Dumper Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file calc_upx.exe

File Type: EXECUTABLE IMAGE

    Summary

       17000 .rsrc
       88000 UPX0
       3D000 UPX1
```

Figure 2.1: DUMPBIN Results of a Calc.exe packed by the UPX packer.

The investigation of binary obfuscation is a lengthy subject in itself and the tools for packing and obfuscation are numerous and readily available. This paper will assume the samples of malware have been stripped of packing or obfuscation prior to examination. Although in a real world implementation the examination of binaries should be combined with a de-obfuscation process.

```
C:\Windows\System32>dumpbin calc.exe
Microsoft (R) COFF/PE Dumper Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file calc.exe

File Type: EXECUTABLE IMAGE

  Summary

    5000 .data
    4000 .reloc
    63000 .rsrc
    53000 .text
```

Figure 2.2: DUMPBIN Results of the Calc.exe binary.

CHAPTER 3

REVERSE ENGINEERING

Two techniques are used today to reverse engineer binaries: static analysis or dynamic analysis. In general, there are advantages and disadvantages to each; one is not necessarily better than the other. Static analysis will be utilized for this paper and the experiment.

3.1 Static Analysis

Static analysis is the process of disassembling or decompiling a binary executable, using a disassembler or decompiler, to generate human readable code [17]. Static analysis allows a reverse engineer to have a more complete view of an executable [22] including code branches and logic. In some cases static analysis can be completed quickly, because the reverse engineer does not need to wait for the program to finish executing or to execute the program over many different iterations with different variables to examine different execution paths. Related, if the binary executable is viewed statically the reverse engineer can safely reverse engineer without the need to quarantine the operating system to prevent the malware from propagating to other computers or contacting C & C servers. But, there are disadvantages to static analysis that should be considered. First and foremost in order to effectively statically analyze a binary executable the reverse engineer should be well-versed in low-level programming languages such as assembly and C. The source code examined for static analysis is usually produced by disassemblers that generate assembly source code or a decompiler that produces C or C-like source code. Decompilers tend to not be as accurate in source code generation as disassemblers for assembly language generation due to programming tricks or compiler optimizations, but even disassemblers can be misled relatively easily. Malware authors are familiar with these limitations and will add bits of code known as "junk bytes" random pieces of code used to obfuscate and “opaque predicates” false branches that always resolve true or false regardless of input. These confuse disassemblers [17] and result in assembly or source code that is deceptive or incorrect. Other drawbacks of static analysis are self-modifying code or polymorphic code, which is commonly found in malware; this code modifies the binary as it executes to change behavior based on the executing environment. Theoretically, static reverse engineering would be able to see this in the disassembly yet in practice this turns out to be difficult because of the many code branches that would be present in the disassembled source code.

3.2 Dynamic Analysis

With dynamic analysis the reverse engineer examines the code executing on a system or emulated system. The reverse engineer views the actual executed instructions, as opposed to static analysis where the reverse engineer has to make an educated guess on which instructions will be executed. The primary drawbacks of dynamic analysis is that malware authors are aware of the fact that analysts execute the malware in quarantined environments or under a debugger and place specific code in the malware to detect those situations [23]. The specific code tests the operating environment to see if it is a virtual machine by testing instruction execution times [24] or using standard Windows calls to detect debuggers [17], if detection is positive the applications will terminate. And finally setting up quarantined environment can be cumbersome and time consuming to tweak the environment iterations to fully test the malware.

3.3 Reverse Engineering Tools

3.3.1 Disassmebler

There are very few disassemblers on the market today, IDA Pro from Hex Rays [25] is the most popular disassembler today and is the disassembler that is used for this paper’s experiments. IDA Pro has support for many different processors, file formats, and available as a executable on Windows, Mac OS X, and the Linux platforms. It also contains a broad library (known as FLIRT) of signatures of standard library functions on Windows, Mac

OS X, and Linux. These library signatures allow a reverse engineer to discover or derive the general behavior of the malware because of the examined executable's use of standard library functions.

3.3.2 Python

For this paper's experiment the Python language is used for automation IDA Pro has deep Python integration which allows automation of the disassembler. In addition Python is somewhat of a *de facto* standard in the reverse engineering community. This is for a couple reasons: first, the language is relatively easy to learn and the structure of scripts are extremely readable. Second, Python as a scripting language allows for low-level access to an operating system which is important for static analysis. Finally because of Python's popularity in the community there are a number of resources and code examples for reverse engineers.

CHAPTER 4
INFORMATION CONTAINED IN BINARY EXECUTABLES

The information contained inside a binary can give a reverse engineer clues about the intended function of the binary and resources it might use. For instance if reverse engineering can derive, through IDA Pro's FLIRT signatures [26] that the WinSocket 2 library is linked to the binary; it can then be deduced that the binary most likely will use a network for data acceptance and/or transmission. With the knowledge the WinSocket 2 library is linked to the binary, imported functions from the WinSocket 2 library, such as `socket()` or `bind()` can be used to gather data about the binary. These imported functions have data passed as arguments, by examining the arguments and following the flow of function calls educated assumptions can be made about the network behavior and the data to be sent on the network by the malware.

```
dsinema$ strings SDBOT05A.exe
...
GetOEMCP
SetStdHandle
LCMapStringA
LCMapStringW
FlushFileBuffers
bot started.
ctcp
raw PRIVMSG $1 :$chr(1)$2-$chr(1)
ping $1 10000 $2 50
udp $1 10000 2048 50
...
```

Figure 4.1: Sample output of the `strings` utility.

4.1 Static Strings

Binaries contain many static strings or static format strings, see Figure 4.1. The static format strings are used by functions such as `sprintf()`, to allow for portions of a static string to contain dynamic information, see Figure 4.2. Because `sprintf()` calls typically precede `send()` calls, a more accurate signature can be produced if the output string of `sprintf()` can be associated with a `send()` call. If that is not possible the strings can still be used in a more general manner. IDA Pro is used to extract string information for the experiment, the usage of the `strings` utility in Figure 4.2 is for illustration. The specifics of how static strings are used for this paper are discussed in detail in Chapter 6.

```
%d, %d : USERID : UNIX : %s
```

Figure 4.2: Example of a format string.

4.2 System Calls

The automation process of this paper looks specifically for system calls that accept or send data on a computer network. For Windows based systems these functions are contained in the WinSocket 2 library found on all modern versions of Windows, and is based on the BSD Sockets library [27]. Examining an Windows binary's header will disclose the imported libraries such as WinSocket 2, which is evidence that Socket functions exist in the targeted binary. Figure 4.3 is a example of a small python script that will list the Windows import address table (IAT). The output of the script at the end of the figure lists the linked libraries for SDBOT05A.exe; WS2_32.dll is the WinSocket 2 library. For this paper's experiment IDA Pro provides this functionality; the script is for illustration.

The remaining part of this chapter examines some of the function calls that are used to determine a binary's network behavior and build signatures. Figure 4.4 from [27] illustrates the flow of data in the WinSocket 2 library, the experiment of this paper follows the data flow of the Socket functions.

```
#!/usr/bin/env python
# Code can be found at http://tinyurl.com/lgg9zmu
import pefile

pe = pefile.PE('SDBOT05A.exe')

for entry in pe.DIRECTORY_ENTRY_IMPORT:
    print entry.dll

Script Output:
dsinema$ ./import-pe.py
            KERNEL32.dll
            ADVAPI32.dll
            SHELL32.dll
            WS2_32.dll
            WININET.dll
```

Figure 4.3: Python Script that Extracts Windows PE Import Address Table Including Execution Results

4.2.1 socket()

The first function in the WinSockets 2 library that could be used for building signatures is the `socket()` call. This call returns a descriptor or socket handle, which is used by other WinSocket 2 functions to send and receive data in that socket. The `socket()` call takes as arguments three integers `afam`, `type`, and `protocol`; `afam` is the address family such as TCP/IP, IPX, or AppleTalk. The second argument is `type`, this value signifies whether the TCP/IP data will be datagrams or a data stream. And finally `protocol` is an integer that matches standard protocol numbers defined by IANA and found on Windows systems in WINDOWS\system32\drivers\etc\protocol. As an example, if the `socket()` function arguments are `socket(AF_INET, SOCK_STREAM, 6)` this would create a TCP/IP socket (AF_INET) of type TCP (SOCK_STREAM) using the protocol of TCP (6).

4.2.2 bind()

The `bind()` function call is used only if the malware is acting as a server. In many cases the malware behaves as both a client and a server, the `bind()` call will help identify which way traffic flows for a socket. The bind() call takes as input a socket handle, a sock_addr structure, and an integer that is the length of the second argument [27]. The socket handle is of interest because it can disclose what type of address family and protocol is being used for the bind() call. The sock_addr structure contains the IP address and port number that the server will listen, of these two pieces of data the port number is the most important. If the port number can be extracted from the bind() call it can be used in signatures to match incoming connections to the malware. Typically malware does not use hardcoded IP addresses or DNS host names but uses dynamically generated DNS names to connect to command and control servers, while the port number will stay consistent. An effective reactive measure against botnets has been to discover the DNS hostname generation algorithm and then to take control of those DNS names, often referred to as sink holing [28]. A discussion of sink holing is outside the scope of this paper, but is mentioned for completeness.

4.2.3 send()

The send() function is utilized if the malware is a server or client. This function takes as arguments the socket handle, the data message in the format of a C string, the length of the data message, and finally some control bits that specify additional behavior. The first two arguments are of most interest, first the socket handle allows the alignment of the data with a socket handle in order to determine the destination of the data. The second argument, the data message, is a pointer to a C string which is used to trace back to a sprintf() function to extract the string. In many cases the string is the best identifier for matching malware, because the string is exposed on computer networks.

4.2.4 sprintf()

The sprintf() function is not part of the sockets library, but it often used prior to a send() call to populate the data message argument for send(). The sprintf() call takes as arguments a C string that will receive the formatted data, a format string, and finally the data for the variables in the format string. This function is important for this paper due to the fact it is used to determine the data being passed to the `send()` function which is directly associated to a socket handle.

4.3 Other Static Data

The other static data of interest in a binary are integers that are passed to functions such as `socket()` or fields in structures such as `sock_addr`. These structures are often stored in the binary in a hexadecimal format and will be discussed more in Chapter 6.

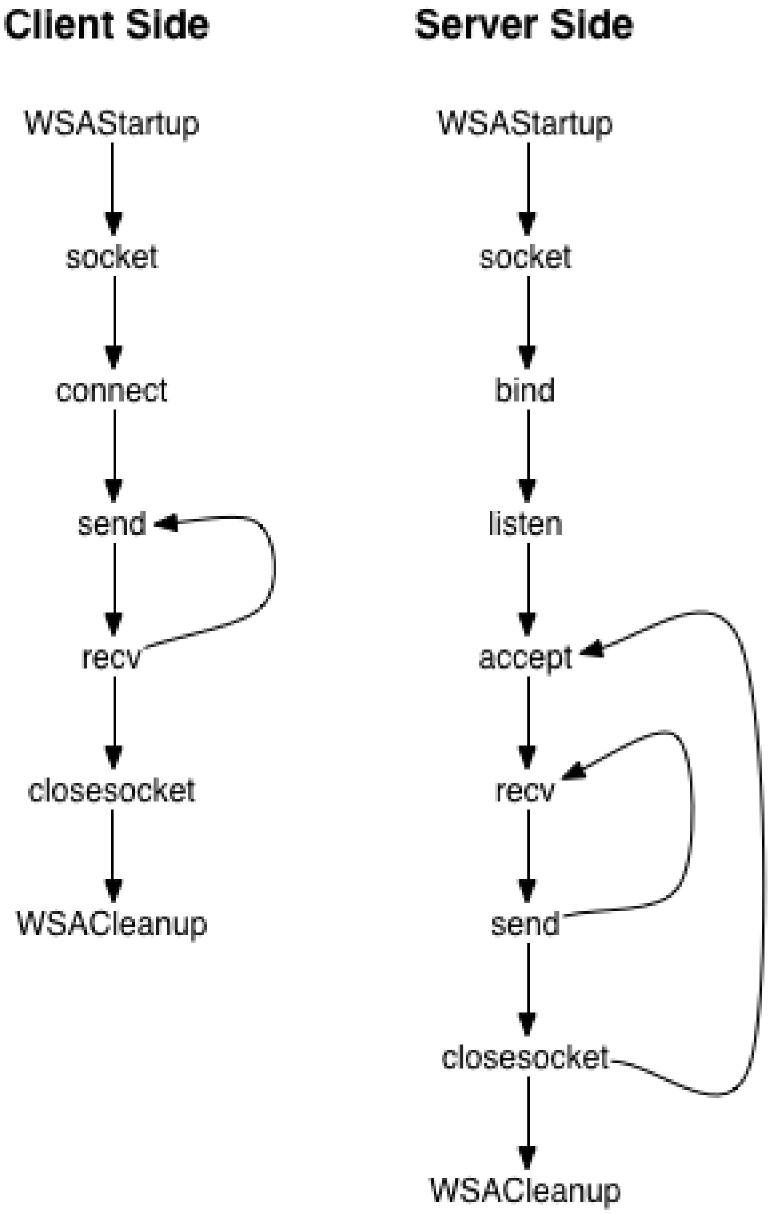

Figure 4.4: Flow of WinSocket 2 function calls. [27]

CHAPTER 5
NETWORK SIGNATURES

5.1 Intrusion Detection Systems

Intrusion detection systems are hardware or software systems deployed on networks or on host systems that attempt to detect suspicious behavior on a network or application by comparing to pre-defined signatures of known malicious network traffic. There are two types of IDSs, host-based intrusion detection systems (HIDS) and network-based intrusion detection systems (NIDS) both systems can be distributed so that data, warnings, and alerts can be aggregated to centralized management servers. This paper is only concerned with NIDS and therefore will only discuss NIDS.

5.1.1 Network-based Intrusion Detection Systems

NIDS is a service that can run on a host or appliance and monitors the traffic on a network segment. The network interface of the NIDS host is configured to a promiscuous mode so that it can view network packets intended for all devices on the same network segment [29]. The signature style of NIDS is typically human readable strings that signify network traffic type, ports used, and strings contained in the data payload of network packet. The signatures are then compared to live traffic by the NIDS parsing the network packets and utilizing keyword pattern matching algorithms to to match packets with known signatures [30]. If the packet matches a signature the operator is notified of the match and can then take appropriate action. This paper we will develop NIDS style signatures from information found in binaries by reverse engineering using static analysis.

5.2 Snort

Snort will be used as the signature reference platform for this paper. Snort is a popular open source IDS that has an active development community, wide deployment and a large number of bundled and community provided signatures [31]. The architecture of Snort is a packet decoder, a detection system and an alert/logging subsystem. Because of the packet decoder the rules can be simplistic, non-binary and user friendly [32]. This paper is not concerned with the alert/logging subsystem or the detection system. To build signatures to be utilized by the alert/logging subsystem or the detection system layers the only need is to assure that the automated signatures are in the proper format for Snort. The packet decoder is of interest because knowledge of the data that will be decoded from network captures assists in determining what type of data is to be found when reverse engineering a binary. The packet decoder is the first subsystem that will handle a network packet once acquired by the packet capture system [31]. A network packet is passed through various decoders for each OSI layer and placed into a data structure internal to Snort for further processing [33]. The parsing will extract information from the packet such as a TCP session (transport) over IP (network) on an Ethernet (data link) network, in addition the payload data will be extracted from the packet. This internal data structure is used to compare against the Snort rules created by the operator. This paper will not compare against the internal data structure, but against Snort rules. This section provides an overview of the Snort system for completeness.

5.3 Anatomy of a Snort Rule

The format of a Snort rule is quite simple, yet it is descriptive enough to build rules that can match a small flow of packets out of a large stream of packets. The rule has essentially two parts, a rule header and a rule option [31]; a sample rule can be seen in Figure 5.1. Although the rule has two major parts, both portions contain many sub-options this paper will only consider a subset of options that directly relate to Socket functions and static strings.

5.3.1 Rule Header

The header of a Snort rule describes where the network packet is coming from, where

```
alert tcp any any -> 10.0.0.1/24 /
(content: "HTTP/1.x 404 Not Found"; msg:"HTTP error";)
```

Figure 5.1: Sample Snort rule.

the network packet is heading, and the transport protocol [31]. The first option of the header section is the rule action which directs Snort what is to be done with a match, this paper is not concerned about the various options for this field as it is outside of the scope. For this paper all automated signatures created have the same rule action, hard coded with the "alert" option from a configurable global setting.

The second option of the header section is protocol, this option informs Snort the type of network protocol used by the network packet. For the generated signatures, this option is based on data that is discovered from reverse engineering and information derived from the `socket()` function arguments.

The third option is IP address, this option is used twice: once for incoming connections and once for outgoing connections. The incoming and outgoing IP addresses are separated by a directional operator which will be discussed later. The IP address sections allows for different formats: the label "any" can be used as a wildcard, a single IP address, IP address ranges in CIDR block format [31], or a variable can be used to define a common network IP range. See Figure 5.2 for an example of a variable for the IP address option. The IP address value is difficult to derive from static reverse engineering, in many cases DNS addresses are used by botnets to allow C & C servers to be moved or the DNS names dynamically generated which complicates detection. In cases where IP addresses are hard coded the data is often hashed or in binary format and part of the `ip_addr` data structure which can be difficult or impossible to statically reverse engineer. For this paper's experiment $HOME_NET and $EXTERNAL_NET will be used for the signatures created, this will allow the operator to globally define IP address ranges in a Snort rules file.

The forth option is the TCP/IP port used by the binary, like IP addresses this option is used twice: once for incoming connections and once for outgoing connections. The format

```
var CORP_NET 10.0.1.0/24
```

Figure 5.2: Sample IP Address Variable.

of the port option is a wildcard label of "any," a single hardcoded port number, or a range of port numbers in the format of a the starting port number in a contiguous range separated by a colon to the ending port number. Figure 5.3 is an example of a range of ports defined in a Snort rule.

```
22:53
```

Figure 5.3: Sample Port Variable.

The fifth and final option of the header section is the directional operator, this operator signifies the flow of the network traffic for the rule. The directional operator has two typographical formats, see Figure 5.4. Note there is not a left arrow for this reason the IP addresses and ports will need to be moved to either side of the right arrow for behavior of a left arrow. The double angle bracket signifies bi-directional traffic [31]. The directional operator is be derived based on the `send()` or `bind()` calls and the associated socket handle.

```
->    <>
```

Figure 5.4: Directional Operators.

5.3.2 Rule Options

The rule options of a Snort rule is not strictly structured, but ordering can improve performance [33]. Because this paper uses a subset of options, ordering for performance is not considered. The options of concern for this paper are `msg`, `flow`, `content`, and `pcre`. The `msg` option is used to give a brief description of the the rule, for this paper's experiment the name of the executable file being examined is used as the value for `msg`. The `flow` option allows Snort to filter out rules that are not traveling in the direction of concerned flow. For example if the flow option is set as `to_server` Snort will only examine packets that are flowing from the client to the server, this value is set according to the use of the Socket function calls `send()` or `bind()` in the source code. The `content` and `pcre` are the core of the options section, this is data that is populated from strings recovered during the reverse engineering process. The `content` option is, in many cases, the exact string recovered. The exception is a format string, the format variable placeholders removed prior to inclusion. The `pcre` option is only used on format strings, the format variable placeholders are replaced with PCRE classes for the appropriate data type. See Figure 5.5 as an example.

```
String format - %d, %d : USERID : UNIX : %s
  PCRE format - \d+, \d+ : USERID :UNIX : \w+
```

Figure 5.5: Example of a format string converted to a PCRE.

CHAPTER 6

BUILDING SIGNATURES

This chapter details the design and architecture used to build network signatures from binaries using static analysis. The process will use the data found in binaries, discussed in Chapter 4 to build signatures based on Snort rule format discussed in Chapter 5.

6.1 Developing Network Signatures with Static Analysis

Section 6.2 will outline the general design for building the network signatures while section 6.3 will detail the implementation of generating signatures. Finally section 6.4 will discuss the method to match the generated signatures with existing Snort rules.

6.2 Design

The design of effective signature generation requires it to be specific enough to identify and extract the necessary data from the binaries, yet at the same time flexible enough to accommodate the different language nuances, compiler optimizations, and developer induced code paths. With these requirements in mind the commonality of the targeted malware binaries is the BSD Sockets library, therefore the design is based upon discovering the Socket function call paths and the flow of data between the Socket functions. The current implementation is specific to Windows binaries, although to extend to other platforms would be relatively simple as most modern operating systems use BSD Sockets as their network stack or at least the basis of their network stack. The design of signature generation relies more on the general features of BSD Sockets and less on the Windows specific implementation of BSD Sockets. For the process of building signatures the data to be discovered in a binary is TCP/IP protocol information, IP addresses, TCP/IP ports, payload data, and finally any static strings in a binary. This data can be found as arguments to the various socket function calls, therefore the implementation of this paper will be to discover call paths from the `send()`, `connect()`, and `bind()` socket library functions to the `socket()` function; collecting as much data as possible from the function arguments and combining the discovered data together to build signatures. The Socket functions are associated to one another by a socket handle, an integer, which is returned by a call to `socket()` and then passed as an argument to the other Socket functions. The binaries targeted for this paper have multiple `socket()` calls as they may act as both a client and a server, make connections to C & C servers, search for infected peers on a local network, or execute local and network based exploits. If there are multiple calls to `socket()` the Socket function calls will need to be associated with the proper socket handle in order to build accurate signatures. Due to the fact it is not possible to statically read socket handles as they are created at runtime and the experiment for this paper only considers static analysis. Socket handles will need to be simulated by mapping call paths from a Socket function to the associated `socket()` call that created the socket handle.

6.3 Implementation

6.3.1 Enumeration of Socket Functions

To map call paths between the Socket functions they first need to be enumerated, beginning with the `socket()` function call locations along with the containing functions. Figure 6.1 is an example of a `socket()` call and the containing function `irc_connect()`

The data from the enumeration will be placed in a custom Python class, discussed in the next section, which has data structures and utility functions to ease building a call tree. This same process is performed for `bind()`, `send()`, `connect()`, and other Socket functions for future use.

6.3.2 NetworkFunction Python Class

NetworkFunction (code listed in Appendix B) is a Python class created for this paper that is used for the enumeration of Socket functions, this class utilizes IDAPython to

```
DWORD WINAPI irc_connect(LPVOID param)
{
    SOCKET sock;
    SOCKADDR_IN ssin;
    IN_ADDR iaddr;
    LPHOSTENT hostent;
    int err, rval;
    char nick[16];
    char *nick1;
    char str[64];
    BYTE spy;
    DWORD id;
    ircs irc;

    ...

    sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Figure 6.1: Example of a `socket()` function call from SDBOT5A.CPP

discover Socket function locations and cross references of the functions. The class accepts, at instantiation, the name of the socket function as the argument which is then passed to the IDAPython's `LocByName()` to recover a linear address of the function in the binary. See Figure 6.2 for an example of the linear address returned by `LocByName()`. With the linear address of the socket function the returned location can be passed to IDAPython's `XrefsTo()` which will return linear address cross references of the passed function as seen in Figure 6.3.

```
Python>hex(LocByName("socket"))
4076aa
```

Figure 6.2: Results for `LocByName()` with socket as the argument

In this example the function `socket` is passed to the `LocByName()` and the cross references of `socket` are returned and stored in `loc`. The `loc` variable is then passed to `XrefsTo()` and the result is enumerated to gather all cross references. This process is then repeated for each of the Socket functions.

```
Python>loc = LocByName("socket")
       for x in XrefsTo(loc):
           print hex(x.frm)
401835
40196b
40675e
406f0d
407004
```

Figure 6.3: Results for `XrefsTo()` with the linear address of socket as the argument

6.3.3 Socket Function Arguments

Following the identification and location of the Socket calls, the next step is to locate the arguments of the Socket functions and extract the argument data if possible. In some cases the data is not accessible from static analysis because the data is created or collected at runtime, in those cases the data will either be disregarded, because it is not germane to the signature, or the data will be populated with a generic placeholder when appropriate. An example would be the `connect()` function's `name` argument which is a `sockaddr` data structure. One of the fields of `sockaddr` is an IP address, extracting that IP address can be very difficult as it could be generated at runtime, is obfuscated, or encrypted. Using the wildcard $HOME_NET or $EXTERNAL_NET (Section 5.3.1) in place of the `sin_addr` feild of `sockaddr` not only resolves the issue of the inability to directly discover the value, but more importantly provide a value that is common with many known SNORT rules. The arguments for a function in assembly are pushed onto the stack prior to the assembly instruction `call`. The arguments are pushed in reverse order; when parsing, this is to be taken into account for the argument algorithm. Figure 6.4 is an example of the `socket(af, type, protocol)` function [27] listed in IDA Pro, the arguments for `socket()` were discussed in detail in section 4.2.1. The example in Figure 6.4 will use the `getArgs()` function from backtrace.py [34] as a utility function to parse the Socket function arguments (code listed in Appendix C).

```
.text:00401965                  push    6               ; protocol
.text:00401967                  push    1               ; type
.text:00401969                  push    2               ; af
.text:0040196B                  call    socket
```

Figure 6.4: IDA Pro Disassembly of a `socket()` Function Call

6.3.4 Enumeration of `bind` and `send` Functions

The same methods used to locate and enumerate the `socket()` function is also used for the `bind()` and `send()` functions.

6.3.5 Bind Function Arguments

The use of `bind()` in a Socket-based application allows the application to accept incoming network connections and provide server functionality. Malware binaries often use `bind()` to connect to other zombies or for executing software exploits, every malware binary examined in the experiment for this paper made at least one `bind()` call. The `bind()` function is defined as `bind(socket, localaddr, addrlen)` [27]. The `socket` argument has been discussed previous and is the same as in the `send()` function. The `localaddr` argument is a `sockaddr` structure, which contains an IP address and port number. The `addrlen` argument is the length of the `localaddr` argument. The port number field of the `sockaddr` structure is of most interest and the process of extracting the port number will be described in this section. The extraction process is similar to that used for `sprintf()`, after identifying the location of the `bind()` call walk back in the assembly until a call to `htons()` is reached. The `htons()` function converts an integer from host byte order to network byte order [27]. Continue walking back from the `htons()` call until an assembly `push` instruction is made, the argument of the `push` call is the port number of `sockaddr`. Figure 6.5 is an example of a `bind()` call from IDA Pro. The port number value is found on line `.text:00401983` it is the argument passed to the `htons()` call on line `.text:0040199C`. If the `push` value is in hexadecimal format it is converted to decimal, for instance `71h` hexadecimal would be converted to `113` decimal. Otherwise if the port number value is not available, if for example it is created at runtime, the port number value of the generated signature is set to any.

```
.text:00401983                push    71h             ; hostshort
.text:00401985                mov     dword ptr [esp+58h+name.sa_family], eax
.text:00401989                mov     [esp+58h+name.sa_family], 2
.text:00401990                mov     dword ptr [esp+58h+name.sa_data+2], eax
.text:00401994                mov     dword ptr [esp+58h+name.sa_data+6], eax
.text:00401998                mov     dword ptr [esp+58h+name.sa_data+0Ah], eax
.text:0040199C                call    htons           ; Call Procedure
.text:004019A1                lea     ecx, [esp+54h+name]
.text:004019A5                push    10h             ; namelen
.text:004019A7                push    ecx             ; name
.text:004019A8                push    esi             ; s
.text:004019A9                mov     word ptr [esp+60h+name.sa_data], ax
.text:004019AE                call    bind            ; Call Procedure
```

Figure 6.5: IDA Pro Disassembly of a bind() Function Call from SDBOT5A.EXE

6.3.6 Send Function Arguments

The send() function is defined as send(socket, buffer, length, flags) [27]. The only argument of concern is the buffer argument, it is passed to the send() function as a reference, because of the limitations of static analysis the pointer reference cannot simply be dereferenced, dereferencing will have to be simulated to access the buffer data. See Figure 6.6 for an example of send() function call in assembly. The lea assembly instruction in this example loads the address of the send() buffer argument, buf, on to the stack. Discovering the buffer argument's variable name the sprintf() call that populates the buf can then be located by walking back in the assembly.

```
.text:00401B3F                push    0               ; flags
.text:00401B41                push    ecx             ; len
.text:00401B42                lea     ecx, [esp+0D08h+buf]
.text:00401B46                push    ecx             ; buf
.text:00401B47                push    esi             ; s
.text:00401B48                call    send
```

Figure 6.6: IDA Pro Disassembly of a send() Function Call

The buffer pointer is usually created with a call to sprintf(), this function allows for the programmer to format a string by placing run-time variables into a static format string. There is access to portions of the format argument of sprintf(), this string can be used for building the signature. See Figure 6.7 for an example of the sprintf() C function.

```
char str[100];
char* pass_value = "my_password_value";
sprintf(str, "PASS %s\r\n", pass_value);
```

Figure 6.7: Example of a sprintf() function call.

The format argument of sprintf() is critical in building signatures for comparison to Snort rules as many of the Snort rules are based upon packet payloads which contain text strings. In Figure 6.7 the format string argument, "PASS %s\r \n", is combined with the arguments that proceed to create a new string stored in the first sprintf() argument in this example it would be str. Through static analysis the format string argument is available, but the latter arguments are typically accessible at runtime. The format string argument generally provides enough information to build an adequate signature for the thesis of this paper. The format string variable is used to populate the content value of the generated signature. In addition because the sprintf() format argument contains variables, the format string argument maps well to regular expressions which allows for the creation of a pcre value for the generated signatures. The pcre option provides more accuracy for Snort rules, but the pcre option does have a drawback of significant performance degradation [29]. Because of the performance issue the pcre option will only be generated if a string contains the sprintf() formatting variables and static text. To simulate dereferencing of the send() buffer argument, begin by walking back the assembly from the send() linear address until a sprintf() call is reached and is within the containing function of send() and sprintf(). Figure 6.8 is the disassembly of the sub_401AF0 function in SDbot5A.exe. This function is a containing function of sprintf() and send().

If the sprintf() function can be located then the arguments are parsed, using the same methods as described previously for socket() [34], focusing on the format argument of sprintf(). When the name of the format argument is discovered it can then be walked back in the assembly to locate the assembly language push instruction for the format

```
.text:00401AF0 ; int __cdecl sub_401AF0(SOCKET s, int, int, char *Source,
int, int, int, char)

...

.text:00401B29                 call    _sprintf         ; Call Procedure

...

.text:00401B48                 call    send             ; Call Procedure
```

Figure 6.8: IDA Pro Disassembly of the function `sub_401AF0` from SDBOT5A.EXE

argument and the value being pushed. This value is typically an offset of the statically defined format string. See Figure 6.9 for an example of the `sprintf()` call in assembly. The offset value is the data used to build the `content` and `pcre` values of the generated signatures.

6.3.7 Build Call Path from send() to socket()

The call path, for reasons discussed earlier, is reconstructed using IDA Pro's cross reference information for `send()` and `socket()`. The NetworkFunction class gathers cross reference information for each of the functions and with NetworkFunction's helper functions cross references are matched. The process works as follows, after locating the `send()` function IDA Pro's `GetFunctionName()` and `LocByName()` [35] is utilized to identify a function containing `send()`, which will be noted as function α. This function α is then compared against `socket()`'s cross references, if the linear address of α is found in the cross references the link is made between `send()` and `socket()`; if a match is not found then the containing function of α, which will be noted as function β is then compared against `socket()`'s cross references. This continues recursively until containing functions are exhausted. It should be noted that matching containing functions might not be foolproof as it relies on `socket()` appearing only once in a function. In practice the malware examined made a `socket()` call only once per containing function, which allows for this method of simulating socket handles to work consistently.

```
.text:00401B23                  push    offset aPassS    ; "PASS %s\r\n"
.text:00401B28                  push    eax              ; Dest
.text:00401B29                  call    _sprintf
```

Figure 6.9: IDA Pro Disassembly of a `sprintf()` Function Call

6.4 Building General Signatures Based on the String Data as a Catch-all

Matching `sprintf()` format strings to a `send()` call is not always possible, due to the method the malware was developed. For instance in SDBot05A.exe the format string arugment is passed into the containing function of `send()` and then passed to `send()`. Although this is traceable when manually reverse engineering the malware, automation proved to be somewhat difficult with the available tools and scripting environment. The work around for this limitation is to list all the strings found in a binary and attach to the `content` option and where applicable the `pcre` option of a generated signature. The generated signature header options would be set as a general header: `alert tcp $HOME_NET -> any $EXTERNAL_NET any`, because `socket()` and therefore the socket handle are unknown. This does have the potential to cause false positives due to common words and terms. The mediation used in this paper is to create an excluded strings list, see Appendix E for a sample list. This list contains strings commonly found in binaries created by compilers or behaviors of a programming language, and are not directly created by the binary's programmer. These words are excluded from consideration for the `content` and `pcre` options in generated signatures. The implementation of this mediation greatly reduced the false positive rate and as a side effect decreased the time required to run the experiment.

6.5 Matching Signatures

This paper used many of the popular and publicly available Snort rules for signature matching (list of rules used are found in Appendix F), it was not limited to solely Snort rules for bots but included all available Snort rules. Based on my research many of the bots contain source code for other exploits and network attacks, while the additional Snort rules might not be specific to a bot, it does allow for more data to properly fingerprint a binary. The matching of generated signatures to Snort rules is based on a weighted score of matching the various parts of a generated signature to a known Snort rule. The header section has the least amount of weight, values could be `tcp` or `$HOME_NET` which are very general and would have a high rate of matching. The header receives more weight if the Snort rules are matched to specific IP addresses, hostnames or port numbers. The header information receives 0.5 point if the values are general and a full 1.0 point if matches are made on specific IP addresses, hostnames, or port numbers. The heavier weight in scoring is placed on the `content` option of the Snort rule. The `content` score is based on letter and word matches and the percentage of the string that is matched. Matching letters prevents common single words from scoring as a match against the target string that could be substantially longer. The threshold is set to 75% match on both letters and words in order to notify the operator that there is a match. The algorithm used to find word matches for this paper is the same algorithm used by Snort, Boyer-Moore [32]. This allows for a partial simulation of real world network traffic and a Snort IDS.

6.6 Results

The implementation of this thesis was tested on a selection of malware bots, the bots were compiled from source code and the binaries were not packed or obfuscated. The two reasons for this choice is first with access to the source code the results found from implementation can be confirmed, second as noted in Chapter 2 because binary obfuscation is not considered in this paper, compiling the bots it can be assured that the binaries are not packed or obfuscated. The results were positive for identifying bots in an automated process when comparing to known Snort rules. The bots analyzed for training were Agobot3, Agobot-phat-glow, and SDBot-5A. For control the Google Chrome browser version 29.0.1547.76 and the Firefox browser version 24.0, these applications also use the WinSock API, were analyzed with the implementation of this thesis.

The summarized findings can be found in Table 6.1, the numbers represent the Snort rules which were returned as matching the examined binary. For instance Agobot3 found 14

Binary Name	Generated Signature Matches to Known Snort Rules								
	IRC	AGOBOT	SDBOT	GTBOT	SPYBOT	HTTP	FTP	MAIL	MISC.
akbot	11	0	1	0	0	278	11	5	13
aspergillus	0	0	0	0	0	0	7	4	2
BioZombie 1.5 Beta	0	0	0	0	0	0	0	0	0
C_15Pub-pre4	6	0	0	0	0	0	5	4	2
C_15Pub.exe	8	0	0	0	0	0	6	4	1
cftmon-dopebot	0	0	0	0	0	0	0	0	0
cftmon	0	0	0	0	0	0	0	0	0
ChodeBase	0	0	0	0	0	0	0	0	0
CYBER-v4.0	10	0	2	1	0	0	14	7	2
CYBER	10	0	2	1	0	0	12	7	2
CYBERBOTv2.2-Stable .m0dd_ownz.DreamWoRK	13	0	2	1	0	0	13	7	2
darkanal	11	0	0	1	0	0	14	7	3
DCI_Bot	0	0	0	0	0	0	0	0	0
GellBot3	7	0	0	0	0	0	0	0	2
GigaBot-DCASS	13	0	4	1	0	292	15	7	5
H-Bot_M0d_3-0_M0dd3d _by_TH_and_Sculay	9	0	0	1	0	286	15	7	5
LiquidBot_FixEd_By_Pr1muZ_anD_Ic3	13	0	4	1	0	287	14	7	5
NESbot_v5	11	0	0	0	0	0	0	2	3
NEW bot by MSIT WIN	8	0	0	0	0	0	1	0	1
rBot-blowSXT	9	0	4	1	0	286	15	7	5
rBot-ciscobawt	9	0	4	1	0	286	15	7	5
rBot-Crackbot_v1.4b	9	0	4	1	0	286	15	7	5
realmbot.exe	13	0	1	0	0	288	15	5	5
Reptilex	13	0	0	1	0	0	2	2	3
rxBot-drx-woopie	13	0	4	1	0	289	15	7	5
spybot	11	0	1	2	2	288	1	0	4
Spybot1.2c-full	12	0	1	2	2	294	0	0	4
Firefox 24.0	0	0	0	0	0	0	0	0	0
Chrome 29.0.1547.76	0	0	0	0	0	0	0	0	0

Table 6.1: Generated Signature Matches to Known Snort Rules

matching signatures for IRC and 18 for AGOBOT, the indication is this binary is a Agobot bot or a derivative of Agobot. It should be noted that many bots analyzed also showed a high rate of other HTTP, FTP, or mail exploits. This finding is consistent with the fact that many bots contain application and protocol exploits that are executed by botmaster from the C & C servers [36]. Therefore the identification of a binary should consider the high rate of exploits in addition to IRC command & control traits. Spybot is an excellent example of the need to examine the overall results, about 95% of the found signatures are HTTP exploits and only about 3% are Spybot, SDBot, and GTBot IRC C & C traits. When looking at overall functionality and purpose of bots [36] the relatively small percentage of bot findings when compared to the percentage of exploit findings is consistent with the behavior and purpose of the bots in a botnet.

CHAPTER 7
FUTURE WORK AND DIRECTIONS

This paper focused on static analysis of binaries, although there was positive success in identifying malicious software it is certainly not fool-proof. To increase the ability to more accurately identify malicious software, dynamic analysis could be employed and the results of both dynamic and static analysis could be combined and compared to increase the confidence of the findings. The open source project Cuckoo Sandbox [37] is based on Python and could be extended to combine the results found in the implementation of this thesis.

Another interesting direction would be to add visualization to the implementation in this thesis. The visualization could be updated dynamically and allow the user to see the genealogy and relationship of bots to other malicious software. Many of the bots examined used the same exploits for HTTP and FTP, closer examination of these relationships could provide more information on more efficient processes to detect the malware.

CHAPTER 8
CONCLUSION

It is estimated that today the average worker has 2.8 connected devices, with projections to this to reach 3.3 by 2014 [38]. Because of the proliferation of devices and accessibility of computer networks the threat of malware has become an increasing concern for businesses and computer users. The objective of this paper is to provide a proactive means to identify malware before it appears on computer networks, thereby allowing for earlier detection and identification. The approach of this paper is to recreate, as much as possible, the network behavior of a malware sample for the purpose of identifying the software as malicious.

The results from the implementation of this paper indicate that malware network behavior can be identified offline through automated static analysis. The malware samples tested against the implementation were identified not only by the IRC traffic, but also by matching of known network exploits. Although not as precise as a network capture with manual examination, the results proved enough information is present from automated static analysis to identify malicious software.

REFERENCES

[1] J. Oltsik. (2013, Sep) Many security professionals don't understand modern malware. [Online]. Available: http://www.networkworld.com/article/2225474/cisco-subnet/many-security-professionals-don-t-understand-modern-malware.html

[2] TrendMicro. (2006, November) Taxonomy of botnet threats. [Online]. Available: http://www.cs.ucsb.edu/~kemm/courses/cs595G/TM06.pdf

[3] J. Leyden. (2009, January) Techwatch weathers ddos extortion attack. [Online]. Available: http://www.theregister.co.uk/2009/01/30/techwatch_ddos/

[4] M. Hines. (2008, January) Botnets: The new political activism. [Online]. Available: http://www.infoworld.com/d/security-central/botnets-new-political-activism-392

[5] R. McMillan. (2010, October) Zeus hackers could steal corporate secrets too. [Online]. Available: http://www.computerworld.com/s/article/9190239/Zeus_hackers_could_steal_corporate_secrets_too?taxonomyId=17&pageNumber=1

[6] R. Lemos. (2008, April) Report: China's botnet problems grows. [Online]. Available: http://www.securityfocus.com/brief/726

[7] J. Leyden. (2011, March) Ddos botnet attacks gold miners and wine makers. [Online]. Available: http://www.theregister.co.uk/2011/03/09/gold_mine_site_botnet/

[8] D. Danchev. (2010, May) Study finds the average price for renting a botnet. [Online]. Available: http://www.zdnet.com/blog/security/study-finds-the-average-price-for-renting-a-botnet/6528

[9] M. S. Sanmeet Kaur, “Automatic attack signature generation systems: A review,” *Security & Privacy*, vol. 11, no. 6, pp. 54–61, November/December 2013.

[10] M. Mohammed, H. Chan, and N. Ventura, “Honeycyber: Automated signature generation for zero-day polymorphic worms,” pp. 1–6, Nov 2008.

[11] Z. Liang and R. Sekar, “Automatic generation of buffer overflow attack signatures: an approach based on program behavior models,” p. 224, Dec 2005.

[12] G. Portokalidis, A. Slowinska, and H. Bos, “Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation,” vol. 40, no. 4, pp. 15–27, 2006.

[13] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, “Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience,” in *IEEE Symposium on Security and Privacy, 2006*. IEEE, 2006, p. 15.

[14] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, “Automatic generation of string signatures for malware detection,” in *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, ser. RAID ’09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 101–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04342-0_6

[15] A. Shabtai, E. Menahem, and Y. Elovici, “F-sign: Automatic, function-based signature generation for malware,” in *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 41, no. 4. IEEE, 2011, pp. 494–508.

[16] G. Tahan, C. Glezer, Y. Elovici, and L. Rokach, “Auto-sign: an automatic signature generator for high-speed malware filtering devices,” *Journal in Computer Virology*, vol. 6, no. 2, pp. 91–103, 2010.

[17] E. Eilam, *Reversing: Secrets of Reverse Engineering*, R. Elliott, Ed. Indianapolis, IN: Wiley Publishing, 2005.

[18] (2011, May) Upx: The ultimate packer for executables. [Online]. Available: http://upx.sourceforge.net/

[19] E. Carrera. (2011, May) pefile is a python module to read and work with pe (portable executable) files. [Online]. Available: http://code.google.com/p/pefile/

[20] (2008, September) pyemu - a python ia-32 emulator. [Online]. Available: http://code.google.com/p/pyemu/

[21] J. Seitz, *Gray Hat Python*, 1st ed. San Francisco, CA: No Starch Press, 2009.

[22] C. Krugel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *Proceedings of the 13th USENIX Security Symposium*, vol. 1. The USENIX Association, August 2004, p. 17.

[23] drupal. (2008, August) Appendix b: Source code - what techniques bots use. [Online]. Available: http://www.honeynet.org/node/56

[24] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *IEEE International Conference on Dependable Systems and Networks with FTCS and DCC, 2008. DSN 2008.*, June 2008, pp. 177–186.

[25] Hex-Rays. (2011, June) Ida pro disassembler - multi-processor, windows hosted disassembler and debugger. [Online]. Available: http://www.hex-rays.com/idapro/

[26] Hex-Rays. (2013, August) Ida f.l.i.r.t. technology: Overview. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/

[27] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Volume III Client-Server Programming and Applications - Windows Sockets Version*, 1st ed., M. Horton, Ed. NJ: Prentice Hall, May 1997.

[28] M. Bowden, *Worm: The First Digital World War*, 1st ed. New York, NY: Alantic Monthly Press, 2011.

[29] R. Alder, J. Dr. Everett F. (Skip) Carter, J. C. Foster, M. Jonkman, R. Marty, and E. Seagren, *Snort IDS and IPS Toolkit*, 1st ed., T. Kohlenberg, Ed. Burlington, MA: Syngress, 2007.

[30] J. Kelly, "An examination of pattern matching algorithms for intrusion detection systems," Master's Thesis, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, August 2006.

[31] I. Sourcefire. (2013, April) Snort :: Home page. [Online]. Available: http://www.snort.org/

[32] M. Roesch, "Snort—lightweight intrusion detection for networks," in *Proceedings of LISA '99: 13th Systems Administration Conference*, USENIX, Ed., November 1999, pp. 229–238.

[33] J. Koziol, *Intrusion Detection with Snort*, 2nd ed. Indianapolis, IN: Sams Publishing, May 2003.

[34] A. Hanel. (2013, August) A script for automated backtracing in ida. [Online]. Available: http://hooked-on-mnemonics.blogspot.com/2013/03/a-script-for-automated-backtracing-in.html

[35] G. Erdélyi, E. Bachaalany, and I. Skochinsky. (2013, April) Idapython - python plugin for interactive disassembler pro. [Online]. Available: https://code.google.com/p/idapython/

[36] P. Barford and V. Yegneswaran, "An inside look at botnets," *Advances in Information Security*, vol. 27, pp. 171–191, 2007.

[37] C. "nex" Guarnieri and C. S. Developers. (2013, April) Cuckoo sandbox - automated malware analysis. [Online]. Available: http://www.cuckoosandbox.org/

[38] cisco. (2012, May) Cisco study: It saying yes to byod. [Online]. Available: http://newsroom.cisco.com/release/854754/Cisco-Study-IT-Saying-Yes-To-BYOD

APPENDICES

Appendix A

SinemaThesis.py

Please see Appendix B and C for information about third party functions used in this script.

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

###########################################################
# I am using backtrace.py for some utility functionality ,
# the script can be found at:
# http://tinyurl.com/m72hr2n
#
# I originally located it from the blog entry of the author,
# Alexander Hanel at:
# http://tinyurl.com/kalyo9f
#
#########################################################

import os
import sys
from string import *
from NetworkFunction import *
from SnortRuleParser import *
from backtrace import *

# Defined Strings
snort_rule_action = "alert"
flowbit_name = lower(rstrip(GetInputFile(), '.exe'))
extern_net = "$EXTERNAL_NET"
home_net = "$HOME_NET"
one_dir = "->"
any_dir = "<>"
any_port = "any"
pcre_string = "\w+"
pcre_digit = "\d+"
pcre_hex = "[0-9A-Fa-f]+"
int_replace = '%d'
str_replace = '%s'
hex_replace = '%x'

# Other structs
regs = ["eax", "ebx", "ecx", "edx", "esi", "ebp"]
tcpip_afam = { 2: 'AF_INET' }
tcpip_types = { 1: 'SOCK_STREAM', 2: 'SOCK_DGRAM', 3: 'SOCK_RAW', 4: 'SOCK_RDM', 5: 'SOCK_SEQPACKET' }
tcpip_protocol = {}
socket_calls = {}
ports = {}
```

```
signatures = []
matching_sigs = {}

# IDAPython enumarated Functions, Structs, and Strings from the .exe
exe_funcs = Functions()
exe_excluded_strs = []
exe_structs = Structs()
exe_strs = Strings()

socket_funcs = NetworkFunction(”socket”)
bind_funcs = NetworkFunction(”bind”)
connect_funcs = NetworkFunction(”connect”)
listen_funcs = NetworkFunction(”listen”)
accept_funcs = NetworkFunction(”accept”)
send_funcs = NetworkFunction(”send”)
recv_funcs = NetworkFunction(”recv”)

snortRuleFileParser = CSnortRuleParser(’/Users/dsinema/Dropbox/Scripts/snort_rules’)

def printStats():
        #
        #
        #       def printStats())
        #
        #       returns:
        #
        #       description:    Print out the SNORT rules that were found to match
        #                                       the examined binary.
        #
        #
        for m in sorted(matching_sigs):
                print ”%d\t%s” % (matching_sigs[m], m)

def addToMatchingSigs(sig):
        #
        #
        #       def addToMatchingSigs(sig)
        #
        #       returns:
        #
        #       description:    Add the passed SNORT rule to the matching_sigs[]
        #                                       structure.
        #
        #
        if sig in matching_sigs:
                matching_sigs[sig.strip()] = matching_sigs[sig] + 1
        else:
                matching_sigs[sig.strip()] = 1

def buildTCPIPProtocolDict():
        #
        #
        #       def buildTCPIPProtocolsDict()
        #
        #       returns:
```

```
        #
        #       description:    Read the protocols file from the host
        #                                       filesystem and add contents to the
        #                                       tcp_protocol{} dictionary
        #
        #
        protocols = open(’/etc/protocols’, ’r’)

        for line in protocols:
                if line.startswith(’#’):
                        pass
                else:
                        tmp = line.split()
                        tcpip_protocol[int(tmp[1])] = tmp[2]

def buildExcludedStrs():
        #
        #
        #       def buildExcludedStrs()
        #
        #       returns:
        #
        #       description:    Read in the exclude text file and import
        #                                       entries into a Python list -> exe_excluded_strs[]
        #
        #
        f = file(”/Users/dsinema/Dropbox/Scripts/ExcludedStrings.txt”, ”r”)
        for e_func in exe_funcs:
                exe_excluded_strs.append(GetFunctionName(e_func))
        for e_str in exe_strs:
                if ”.dll” in str(e_str):
                        exe_excluded_strs.append(strip(str(e_str)))
                elif ”.DLL” in str(e_str):
                        exe_excluded_strs.append(strip(str(e_str)))
                if str(e_str)[0] == ’?’:
                        exe_excluded_strs.append(strip(str(e_str)))
        for f_line in f:
                exe_excluded_strs.append(strip(f_line))

def buildSocketCalls():
        #
        #
        #       def buildSocketCalls()
        #
        #       returns:
        #
        #       description:    This builds the socket_func_struct [] Python list
        #                                       object. This list is used to keep track of the
        #                                       socket calls in the examined binary.
        #
        #
        if not socket_calls:
                socket_func_struct = []
                for crefs in socket_funcs.getcodexrefs():
                        socket_func_struct.append(GetFuncOffset(crefs))
                        socket_func_struct .append(findSocketArgs(crefs))
                        socket_calls [GetFunctionName(crefs)] = socket_func_struct
                        socket_func_struct  = []

def BoyerMooreHorspool(pattern, text):
    #
        # http://tinyurl.com/mabdzdd
        #
        # bmh.py
        #
        # An implementation of Boyer-Moore-Horspool string searching.
        #
        # This code is Public Domain.
        #
        m = len(pattern)
        n = len(text)
        if  m > n: return -1
        skip = []
        for k in range(256): skip.append(m)
        for k in range(m - 1): skip[ord(pattern[k])]  = m - k - 1
        skip = tuple(skip)
        k = m - 1
        while k < n:
                j  = m - 1; i = k
                while j  >= 0 and text[i] == pattern[j]:
                        j  -= 1; i -= 1
                if  j  == -1: return i + 1
                k += skip[ord(text[k])]
        return -1

def findSocketHandle(func):
        #
        #
        #       def findSocketHandle(func)
        #
        #       returns:        func_frm/func_tmp       - ea value
        #
        #       description:    Find a socket()  call  related  to  this
        #                                       function.  This function  is  recursive
        #                                       and will  walk back until  it  is  unable
        #                                       to  find  cross  references  for  containing
        #                                       functions.
        #
        #
        func_tmp = ""
        count = 0
        hit_main = 0
        func_name = GetFunctionName(func)
        if  socket_funcs.matchfuncxrefs(func_name):
                return func_name
        else:
                for x in XrefsTo(func):
                        count = count + 1
                        if XrefTypeName(x.type) == 'Data_Offset' and SegName(x.frm) != '.text':              # exclude refs to
                              the non-text segments
                pass
            else:
                func_tmp = GetFunctionName(x.frm)
                if func_tmp == func_name:
                    break
                if socket_funcs.matchfuncxrefs(func_tmp):
                    return func_tmp
                else:
                    func_frm = findSocketHandle(LocByName(func_tmp))
                    if socket_funcs.matchfuncxrefs(func_frm):
                        return func_frm

def findRegValue(address, reg):
    #
    #
    #    def findRegValue(address, reg)
    #
    #    returns:        val     − string value
    #
    #    description:    From the address passed in look for the
    #                    value of the register passed in. This looks
    #                    specifically for the assembly instructions
    #                    mov, xor, pop as these are typically used
    #                    to populate the registers.
    #
    #
    regFound = 0
    currentAddr = PrevHead(address, minea=0)
    while regFound == 0:
        if ’mov’ in GetDisasm(currentAddr) or ’lea’ in GetDisasm(currentAddr):
            opt = GetOpnd(currentAddr,0)
            if str(opt) == str(reg):
                val = str(GetOpnd(currentAddr,1))
                if val.endswith(’h’):
                    val = int(val[:−1], 16)
                return val
        elif ’xor’ in GetDisasm(currentAddr):
            opt = GetOpnd(currentAddr,0)
            if str(opt) == str(reg):
                return ”0”
        elif ’pop’ in GetDisasm(currentAddr):
            opt = GetOpnd(currentAddr,0)
            if str(opt) == str(reg):
                currentAddr = PrevHead(currentAddr, minea=0)
                val = str(GetOpnd(currentAddr,0))
                return val
        currentAddr = PrevHead(currentAddr, minea=0)

def funcArgs(address, num):
    #
    #
    #    def funcArgs(address, num)
    #
    #    returns:        addr    − ea value
    #    value   − String value (NOTE: every value is cast to a string)
```

```
        #
        #       description:    Locate and return the args for the function located
        #                                       at the passed address
        #
        #
        s = Backtrace()
        addr, value = s.getArgs(address, num)
        if value in regs:
                if value.endswith('h'):
                        value = str(int(value[:-1], 16))
                else:
                        value = findRegValue(addr, value)
        elif value.endswith('h'):
                value = str(int(value[:-1], 16))
        return addr, str(value)

def findSocketArgs(address):
        #
        #
        #       def findSocketArgs(address)
        #
        #       returns:        arg1, arg2, arg3        - strings
        #
        #       description:    Find the arguments for the socket()
        #                                       function call. This function also maps
        #                                       the numerical values for the arguments to
        #                                       the protocol names and family names.
        #                                       Example: 6 -> TCP
        #
        #
        addr, arg1 = funcArgs(address, 1)
        addr, arg2 = funcArgs(address, 2)
        addr, arg3 = funcArgs(address, 3)
        if arg1.isdigit():
                if int(arg1) in tcpip_afam:
                        arg1 = tcpip_afam[int(arg1)]
        else:
                pass

        if arg2.isdigit():
                if int(arg2) in tcpip_types:
                        arg2 = tcpip_types[int(arg2)]
        else:
                if arg3.isdigit():
                        if int(arg3) in tcpip_protocol:
                                if int(arg3) == 6:
                                        arg2 = tcpip_types[1]
                                elif int(arg3) == 17:
                                        arg2 = tcpip_types[2]

        if arg3.isdigit():
                if int(arg3) in tcpip_protocol:
                        arg3 = tcpip_protocol[int(arg3)]
        else:
                pass
        return arg1, arg2, arg3

def sprintfArgValues(address, count):
        #
        #
        #       def sprintfArgValues(address, count)
        #
        #       returns:        tmp     -  string
        #
        #       description:    Find the arguments values for the sprintf()
        #                                       function call.
        #
        #
        tmp = []
        x = count
        currentAddr = PrevHead(address, minea=0)
        while (x > 0):
                if 'push' in GetDisasm(currentAddr):
                        opt = GetOpnd(currentAddr,0)
                        if 'offset' in opt:
                                off_tmp = opt.lstrip('offset').strip()
                                off_val = GetDisasm(LocByName(off_tmp))
                                if 'db' in off_val:
                                        off_val = off_val.lstrip('db').strip()
                                        if off_val.endswith('h'):
                                                off_val = str(int(off_val[:-1], 16))
                                                tmp.append(off_val)
                                x = x - 1
                        else:
                                if opt.endswith('h'):
                                        opt = str(int(opt[:-1], 16))
                                tmp.append(opt)
                                x = x - 1
                currentAddr = PrevHead(currentAddr, minea=0)
        return tmp

def pcreSprintfFormat(format_struct):
        #
        #
        #       def pcreSprintfFormat(format_struct)
        #
        #       returns:        retval  -  string
        #
        #       description:    Replace the printf format variables
        #                                       in the string with PCRE compatible
        #                                       variables.
        #
        #
        carriage_return = '\\r'
        new_line = '\\n'
        retval = ""
        for key in format_struct.keys():
                tmp_list = format_struct[key]
                count = len(tmp_list)
        key = key.replace(carriage_return, "").replace(new_line, "") # strip off any returns and new lines
        if key == '%s':
            retval = None
        else:
            retval = retval + "/"
            for i, c in enumerate(key):
                if c == '%':
                    var = str(tmp_list.pop())
                    if key[i+1] == 'd':
                        if var in regs:
                            retval = retval + pcre_digit
                        else:
                            retval = retval + var
                    elif key[i+1] == 's':
                        if var in regs:
                            retval = retval + pcre_string
                        else:
                            retval = retval + var
                    elif key[i+1] == 'x':
                        if var in regs:
                            retval = retval + pcre_hex
                        else:
                            retval = retval + var
                elif key[i-1] == '%':
                    pass
                else:
                    retval = retval + c
    if retval:
        if retval.count(":"):
            retval = retval.replace(":", "")
        retval = retval + "/"
    return retval

def sprintfFormatArg(address):
    #
    #
    #       def sprintfFormatArg(address)
    #
    #       returns:        retval  - string
    #
    #       description:    Remove all the printf format
    #                               varaibles.
    #
    #
    sprintf_dict = {}
    retval = ""
    addr, value = funcArgs(address, 2)
    if 'offset' in value:
        tmp_str = GetDisasm(addr)
        tmp_dict = tmp_str.split(';')
        value = tmp_dict[1]
        value = value.strip()
        value = value.strip('".')
        sprintf_dict[value] = sprintfArgValues(addr, value.count('%'))
        retval = pcreSprintfFormat(sprintf_dict)
```

```
                return retval
        else:
                return None

def findSendData(address):
        #
        #
        #       def findSendData(address)
        #
        #       returns:        formatArg       -  string
        #
        #       description:    Walks back from the send() function
        #                       call to find a sprintf() call. Looks for
        #                       first the assembly instruction call and
        #                       then for sprintf. This function will not
        #                       walk past the containing function.
        #
        #
        func_boundry = LocByName(GetFunctionName(address))
        regFound = 0
        currentAddr = PrevHead(address, minea=0)
        while regFound == 0:
                if currentAddr == func_boundry:
                        return ""
                else:
                        if 'call' in GetDisasm(currentAddr):
                                opt = GetOpnd(currentAddr,0)
                                if 'sprintf' in opt:
                                        formatArg = sprintfFormatArg(currentAddr)
                                        return formatArg
                currentAddr = PrevHead(currentAddr, minea=0)

def commentFromPRCE(string):
        #
        #
        #       def commentFromPRCE(string)
        #
        #       returns:        string  -  string
        #
        #       description:    Utility function that strips
        #                       the PCRE string of the PCRE
        #                       variables.
        #
        #
        if string.count(":"):
                string = string.replace(":", "")
        if string.count(pcre_string):
                string = string.replace(pcre_string, "")
        if string.count(pcre_digit):
                string = string.replace(pcre_digit, "")
        if string.count(pcre_hex):
                string = string.replace(pcre_hex, "")
        return string

def findPortFromBind(address):
        #
        #
        #        def findPortFromBind(address)
        #
        #        returns:          tmp     -  string
        #
        #        description:     Walks back from the bind() function
        #                                             call to locate the port argument. This
        #                                             works similar to the findSendData function
        #                                             above.
        #
        #
        htons_found = 0
        push_found = 0
        currentAddr = PrevHead(address, minea=0)
        while htons_found == 0:
                if 'call' in GetDisasm(currentAddr):
                        htons_found = 1
                        while push_found == 0:
                            if 'push' in GetDisasm(currentAddr):
                                value = str(GetOpnd(currentAddr,0))
                                if value.endswith('h'):
                                    value = str(int(value[:-1], 16))
                                    return value
                                elif value in regs:
                                    tmp = findRegValue(currentAddr, value)
                                    if tmp.endswith('h'):
                                        tmp = str(int(tmp[:-1], 16))
                                        return tmp
                                    else:
                                        return "any"
                                else:
                                    return "any"
                            currentAddr = PrevHead(currentAddr, minea=0)
                currentAddr = PrevHead(currentAddr, minea=0)

def buildPortInformation():
        #
        #
        #        def buildPortInformation()
        #
        #        returns:
        #
        #        description:     maps a port to a bind() function call
        #
        #
        for brefs in bind_funcs.getcodexrefs():
                port = findPortFromBind(brefs)
                ports[GetFunctionName(brefs)] = port

def buildSnortSignatures():
        #
        #
        #        def buildSnortSignatures()
        #
```

```
#       returns:
#
#       description:    The meat of the script this is the
#                                       function that creates the signatures.
#
#
#
snort_rule = ""
buildPortInformation()
for screfs in send_funcs.getcodexrefs():
        pcre_tmp = findSendData(screfs)
        if pcre_tmp:
                if bind_funcs.matchfunc(screfs):
                        if bind_funcs.getfunc(screfs) in socket_calls:
                                if bind_funcs.getfunc(screfs) in ports:
                                        snort_rule = snort_rule_action + "␣" + lower(socket_calls[bind_funcs.getfunc(screfs
                                              )][1][2])  + "␣" + home_net + "␣" + ports[bind_funcs.getfunc(screfs)] + "␣"
                                              + one_dir + "␣"+ extern_net + "␣" + any_port + "␣("
                                        snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + 'flowbits:set,score6_'
                                               + flowbit_name + ';␣'
                                        if 'tcp' in snort_rule:
                                                snort_rule = snort_rule +'flow:from_server,␣established;␣'
                                        elif 'udp' in snort_rule:
                                                snort_rule = snort_rule +'flow:from_server,␣stateless;␣'
                                        if pcre_tmp:
                                                snort_rule = snort_rule + 'content:"' + commentFromPRCE(pcre_tmp) + '
                                                      ";␣'
                                                snort_rule = snort_rule + 'pcre:"' + pcre_tmp + '";'
                                else:
                                        snort_rule = snort_rule_action + "␣" + lower(socket_calls[bind_funcs.getfunc(screfs
                                              )][1][2])  + "␣" + home_net + "␣" + any_port + "␣" + one_dir + "␣"+
                                              extern_net + "␣" + any_port + "␣("
                                        snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + 'flowbits:set,score5_'
                                               + flowbit_name + ';␣'
                                        if pcre_tmp:
                                                snort_rule = snort_rule + 'content:"' + commentFromPRCE(pcre_tmp) + '
                                                      ";␣'
                                                snort_rule = snort_rule + 'pcre:"' + pcre_tmp + '";'
                                snort_rule = snort_rule + ')'
                                signatures.append(snort_rule)
                else:
                        if findSocketHandle(screfs) in socket_calls:
                                if findSocketHandle(screfs) in ports:
                                        snort_rule = snort_rule_action + "␣" + lower(socket_calls[findSocketHandle(screfs)
                                               ][1][2])  + "␣" + home_net + "␣" + ports[findSocketHandle(screfs)] + "␣" +
                                              one_dir + "␣"+ extern_net + "␣" + any_port + "␣("
                                        snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + 'flowbits:set,score4_'
                                               + flowbit_name + ';␣'
                                        if 'tcp' in snort_rule:
                                                snort_rule = snort_rule +'flow:from_server,␣established;␣'
                                        elif 'udp' in snort_rule:
                                                snort_rule = snort_rule +'flow:from_server,␣stateless;␣'
                                        if pcre_tmp:
                                                snort_rule = snort_rule + 'content:"' + commentFromPRCE(pcre_tmp) + '
                                                      ";␣'
```

```
                            snort_rule = snort_rule + 'pcre:"' + pcre_tmp + '";'
                    else:
                        snort_rule = snort_rule_action + "␣" + lower(socket_calls[findSocketHandle(screfs)
                                ][1][2]) + "␣" + home_net + "␣" + any_port + "␣" + one_dir + "␣"+
                               extern_net + "␣" + any_port + "␣("
                        snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + 'flowbits:set,score3_'
                                + flowbit_name + ';␣'
                        if pcre_tmp:
                            snort_rule = snort_rule + 'content:"' + commentFromPRCE(pcre_tmp) + '
                                  ";␣'
                            snort_rule = snort_rule + 'pcre:"' + pcre_tmp + '";'
                    snort_rule = snort_rule + '␣)'
                    signatures.append(snort_rule)

def formatStringValue(string):
    #
    #
    #   def buildSnortSignatures()
    #
    #   returns:
    #
    #   description:    The meat of the script this is the
    #                           function that creates the signatures.
    #
    #
    #
    carriage_return = '\r'
    new_line = '\n'
    retval = ""
    tmp_str = ""
    tmp_str = string.replace('\r', "").replace('\n', "")
    if tmp_str.count('%'):
        tmp_list = tmp_str.split()
        for i in range(0,len(tmp_list)):
            retval = retval + tmp_list[i].replace(int_replace, "").replace(str_replace, "")
            if i < (len(tmp_list) - 1):
                retval = retval + "␣"
    else:
        retval = tmp_str
    return retval

def pcreFromComment(string):
    carriage_return = '\r'
    new_line = '\n'
    retval = ""
    tmp_str = ""
    tmp_str = string.replace('\r', "").replace('\n', "")
    if tmp_str.count('%'):
        retval = retval + "/"
        tmp_list = tmp_str.split()
        for i in range(0,len(tmp_list)):
            retval = retval + tmp_list[i].replace(int_replace, pcre_digit).replace(str_replace, pcre_string).replace(
                   hex_replace, pcre_hex).replace(":", "")
            if i < (len(tmp_list) - 1):
                retval = retval + "␣"
                retval = retval + "/"
        else:
                retval = ""
        return retval

def stringSnortRules():
                snort_rule = ""
                for s in exe_strs:
                        if formatStringValue(str(s).strip()) not in exe_excluded_strs:
                                for strrefs in XrefsTo(s.ea):
                                        if SegName(strrefs.frm) == '.text':
                                                if findSocketHandle(strrefs.frm) in socket_calls:
                                                        if len(formatStringValue(str(s))) > 0:
                                                                snort_rule = snort_rule_action + "␣" + lower(socket_calls[
                                                                      findSocketHandle(strrefs.frm) ][1][2]) + "␣" + home_net + "
                                                                      ␣" + any_port + "␣" + one_dir + "␣"+ extern_net + "␣" +
                                                                      any_port + "␣("
                                                                snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + '
                                                                      flowbits:set,score2_' + flowbit_name + ';␣'
                                                                snort_rule = snort_rule + 'content:"' + formatStringValue(str(s))
                                                                      + '";'
                                                                if len(pcreFromComment(str(s))) > 0:
                                                                        snort_rule = snort_rule + '␣pcre:"' + pcreFromComment(
                                                                              str(s)) + '";'
                                                else:
                                                        if len(formatStringValue(str(s))) > 0:
                                                                snort_rule = snort_rule_action + "␣" + "tcp" + "␣" + home_net +
                                                                      "␣" + any_port + "␣" + one_dir + "␣"+ extern_net + "␣" +
                                                                      any_port + "␣("
                                                                snort_rule = snort_rule + 'msg:"' + GetInputFile() + '";␣' + '
                                                                      flowbits:set,score1_' + flowbit_name + ';␣'
                                                                snort_rule = snort_rule + 'content:"' + formatStringValue(str(s))
                                                                      + '";'
                                                                if len(pcreFromComment(str(s))) > 0:
                                                                        snort_rule = snort_rule + '␣pcre:"' + pcreFromComment(
                                                                              str(s)) + '";'
                                        if len(snort_rule) > 0:
                                                snort_rule = snort_rule + "␣)"
                        if len(snort_rule) > 0:
                                signatures.append(snort_rule)
                        snort_rule = ""

def findSignatureMatches(sigs, rules_lib):
    for sig in sigs.rules:
                sig_word_total = len(sig.contents.split())
                 sig_letter_total = len(sig.contents)
                for rule in rules_lib.rules:
                        rule_word_total = len(rule.contents.split())
                         rule_letter_total = len(rule.contents)
                        count = 0
                        if len(rule.contents) > 0:
                                if sig.contents in string.punctuation:
                                        pass                                                                    # This deals with
                                              strings are punctuation (.,:;)
                                else:
```

```
                    s = BoyerMooreHorspool(sig.contents, rule.contents)
                    if s > -1:
                        if rule.hdr_proto == sig.hdr_proto:
                            count = count + 1
                        for port_rule in rule.hdr_ports:
                            for sig_port in sig.hdr_ports:
                                if sig_port == port_rule:
                                    #print "Port: " + sig_port
                                    if sig_port != 'any':
                                        count = count + 1
                                    else:
                                        count = count + 0.5
                        if ((sig_word_total/rule_word_total) * 100) > 75:
                            if (( sig_letter_total / rule_letter_total ) * 100) > 75:
                                addToMatchingSigs(rule.msg)
                    else:
                        pass

def buildSnortRule():
    buildTCPIPProtocolDict()
    buildSocketCalls()
    buildExcludedStrs()
    buildSnortSignatures()
    stringSnortRules()
    signatureRuleParser = CSnortRuleParser(signatures)
    findSignatureMatches(signatureRuleParser, snortRuleFileParser)
    printStats()


buildSnortRule()
```

Appendix B

NetworkFunction.py

```
#!/usr/bin/env python
from idaapi import *
from idc import *
from idautils import *

class NetworkFunction:
        def __init__(self, name=None):
                self.name = name
                self.address = None
                self.codexrefs = []
                self.funcxrefs = []
                self.funcxrefsaddr = []
                self.setaddress()
                self.setfuncxrefs()
                self.setfuncxrefsaddr()
                self.setcodexrefsTo()

######################################################################
#       Name:                           setaddress()
#       input variables:        Nothing
#
#       Returns:                        Nothing
#
#       Description:            This function sets the address of
#                                               the function name.
######################################################################

        def setaddress(self):
                if self.name:
                        self.address = LocByName(self.name)
                else:
                        self.address = None

######################################################################
#       Name:                           setxrefs()
#       input variables:        Nothing
#
#       Returns:                        Nothing
#
#       Description:            This function sets the self.xrefs list
#                                               to xrefs for the NetworkFunction. This function
#                                               calls the XrefsTo() IDAPython function call
#                                               to walk the xrefs.
######################################################################
```

```
        def setfuncxrefs(self):
                for xref in XrefsTo(self.address, 0):
                        funcName = GetFunctionName(xref.frm)
                        if self.funcxrefs.count(funcName) < 1:
                                self.funcxrefs.append(funcName)
                #self.funcxrefs.sort()

        def setfuncxrefsaddr(self):
                #print hex(self.address)
                for xref in XrefsTo(self.address, 0):
                        #print hex(xref.frm)
                        func = get_func(xref.frm)
                        if func:
                                #print hex(func.startEA)
                                if self.funcxrefsaddr.count(func.startEA) < 1:
                                        #print "func: %s" % hex(func)
                                        self.funcxrefsaddr.append(func.startEA)
                #self.funcxrefs.sort()

        def setcodexrefsTo(self):
                for xref in CodeRefsTo(self.address, 1):
                        #print xref
                        #print hex(xref)
                        if xref:
                                self.codexrefs.append(xref)
                #self.codexrefs.sort()

####################################################################################
#       Name:                           getname()
#       input variables:        Nothing
#
#       Returns:                        name of NetworkFunction object
#
#       Description:            This function returns the name of
#                                               NetworkFunction
####################################################################################

        def getname(self):
                if self.name:
                        return self.name
                else:
                        return None

####################################################################################
#       Name:                           getaddress()
#       input variables:        Nothing
#
#       Returns:                        address of NetworkFunction object
#
#       Description:            This function returns the address of the
#                                               NetworkFunction
####################################################################################

        def getaddress(self):
                if self.address:
```

```
            return self .address
        else :
            return None

##################################################################
#       Name:                           getxrefs ()
#       input variables :        Nothing
#
#       Returns:                         list  of  xrefs  for  function
#
#       Description:             This function returns a  list  of address
#                                               of xrefs  to  NetworkFunction
##################################################################

    def getcodexrefs( self ):
        return self .codexrefs

    def getfuncxrefs( self , num=None):
        if num:
            return self . funcxrefs [num]
        else :
            return self . funcxrefs

    def getfuncxrefsaddr( self , num=None):
        if num:
            return self . funcxrefsaddr[num]
        else :
            return self . funcxrefsaddr

    def getfunc( self , addr=None):
        if addr:
            for s in self .codexrefs:
                #print ”s: %s” % hex(s)
                if GetFunctionName(s) == GetFunctionName(addr):
                    return GetFunctionName(s)
                else :
                    return None
        else :
            return None


##################################################################
#       Name:                           addxrefs()
#       input variables :        address
#
#       Returns:                         Boolean
#
#       Description:             This function adds to self . xrefs ,
#                                               this allows future proofing.  The function
#                                               returns True if the address was added
#                                               successfully , and False if not.
##################################################################

    def addxrefs( self , address=None):
        if address:
```

```
                        func_name = GetFunctionName(address)
                        self.xrefs.append(address)                                      # add item to list
                        self.xrefs.sort()                                                       # sort the list after ...
                        if self.funcxrefs.count(func_name) < 1:
                                self.funcxrefs.append(func_name)
                                self.funcxrefs.sort()
                                                                                # error out in future   ** TODO **

        def addstring(self, string=None):
                if string:
                        self.nearstrings.append(string)

#################################################################
#       Name:                           matchxrefs()
#       input variables:        address
#
#       Returns:                        Boolean
#
#       Description:            This function searches the self.xrefs to
#                                               find a match for the given address.
#                                               Returns True if found otherwise False.
#################################################################

        def matchxref(self, address=None):
                if address:                                             # if address is valid, search
                        for xref in self.xrefs:
                                #print xref
                                if str(address) == str(xref):
                                        return True
                        return False # Looped through list and address was not found
                else:
                        return False

#################################################################
#       Name:                           matchname()
#       input variables:        name
#
#       Returns:                        Boolean
#
#       Description:            This function is a utility function to
#                                               match the name of the NetworkFunction
#                                               object with the given name.
#################################################################

        def matchname(self, name=None):
                if name == self.name:
                        return True
                else:
                        return False

#################################################################
#       Name:                           matchaddress()
#       input variables:        address
#
#       Returns:                        Boolean
```

```
#
#       Description:            This function is a utility function to match
#                                               the address of the NetworkFunction object
#                                               with the given address.
#################################################################

    def matchaddress(self, address=None):
        if str(address) == str(self.address):
            return True
        else:
            return False

    def matchfuncxrefs(self, func_name=None):
        if func_name:
            if func_name in self.funcxrefs:
                return True
            return False
        else:
            return False

    def matchfunc(self, addr=None):
        if addr:
            for s in self.codexrefs:
                #print ”s: %s” % hex(s)
                if GetFunctionName(s) == GetFunctionName(addr):
                    return True
                else:
                    return False
        else:
            return False
```

Appendix C

Backtrace.py

Backtrace.py was used for some utility functionality, the entire script can be found at `https://bitbucket.org/Alexander_Hanel/backtrace`. This appendix will only include the function `getArgs()` that was used from backtrace.py.

```
def getArgs(self, address, count):
    'get␣specified␣(by␣count)␣argument␣and␣address'
    pushcount = 0
    instructionMax = 10 + count
    currAddress = PrevHead(address,minea=0)
    while pushcount <= count and instructionMax != 0:
        if 'push' in GetDisasm(currAddress):
            pushcount += 1
            if pushcount == count:
                return currAddress, GetOpnd(currAddress,0)
        instructionMax -= 1
        currAddress = PrevHead(currAddress,minea=0)
    return None, None
```

Appendix D

Boyer-Moore-Horspool Function

A public domain implementation of the Boyer-Moore-Horspool algorithm found at `http://code.activestate.com/recipes/117223-boyer-moore-horspool-string-searching/` was used for the Boyer-Moore algorithm. The function is listed below for completeness.

```
def BoyerMooreHorspool(pattern, text):
    # An implementation of Boyer−Moore−Horspool string searching.
    #
    # This code is Public Domain.
    #
    m = len(pattern)
    n = len(text)
    if m > n: return −1
    skip = []
    for k in range(256): skip.append(m)
    for k in range(m − 1): skip[ord(pattern[k])] = m − k − 1
    skip = tuple(skip)
    k = m − 1
    while k < n:
        j = m − 1; i = k
        while j >= 0 and text[i] == pattern[j]:
            j −= 1; i −= 1
        if j == −1: return i + 1
        k += skip[ord(text[k])]
    return −1
```

Appendix E

Excluded Strings List

This string list is used to cut down the false positives when searching for strings in a binary to use for the Snort rule `content` option. Most of the strings are compiler generated or used for language specifics used internally by the binary and runtime environment and are typically not exposed on a network.

```
kernel32
GetProcAddress
LoadLibraryA
ExitThread
msvcrt
system
IsProcessorFeaturePresent
KERNEL32
runtime error
RegisterServiceProcess
Microsoft Visual C++ Runtime Library
Runtime Error!\n\nProgram:
Runtime Error!Program:
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
InternetGetConnectedStateEx
IcmpSendEcho
```

```
IcmpCloseHandle

IcmpCreateFile

0123456789abcdefghijklmnopqrstuvwxyz

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ

0123456789+- .*#hlL^ztjqZw’&@I

Process32Next

Process32First

CreateToolhelp32Snapshot

RegisterServiceProcess

runtime error

abnormal program termination

main thread

ActivateKeyboardLayout

AdjustWindowRect

AdjustWindowRectEx

AlignRects

AllowForegroundActivation

AllowSetForegroundWindow

AnimateWindow

AnyPopup

AppendMenuA

AppendMenuW

ArrangeIconicWindows

AttachThreadInput

...
```

Appendix F

Snort Rules Used for the Experiment

The Snort rules used in this experiment can be found:

F.1 Snort Community rules:

http://www.snort.org/snort-rules/

app-detect.rules

attack-responses.rules

backdoor.rules

bad-traffic.rules

blacklist.rules

botnet-cnc.rules

browser-chrome.rules

browser-firefox.rules

browser-ie.rules

browser-other.rules

browser-plugins.rules

browser-webkit.rules

chat.rules

content-replace.rules

ddos.rules

deleted.rules

dns.rules

dos.rules

experimental.rules

exploit-kit.rules

exploit.rules

file-executable.rules

file-flash.rules

file-identify.rules

file-image.rules

file-java.rules

file-multimedia.rules

file-office.rules

file-other.rules

file-pdf.rules

finger.rules

ftp.rules

icmp-info.rules

icmp.rules

imap.rules

indicator-compromise.rules

indicator-obfuscation.rules

indicator-scan.rules

indicator-shellcode.rules

info.rules

local.rules

malware-backdoor.rules

malware-cnc.rules

malware-other.rules

malware-tools.rules

misc.rules

multimedia.rules

mysql.rules

netbios.rules

nntp.rules

oracle.rules

os-linux.rules

os-mobile.rules

os-other.rules

os-solaris.rules

os-windows.rules

other-ids.rules

p2p.rules

phishing-spam.rules

policy-multimedia.rules

policy-other.rules

policy-social.rules

policy-spam.rules

policy.rules

pop2.rules

pop3.rules

protocol-dns.rules

protocol-finger.rules

protocol-ftp.rules

protocol-icmp.rules

protocol-imap.rules

protocol-nntp.rules

protocol-pop.rules

protocol-rpc.rules

protocol-scada.rules

protocol-services.rules

protocol-snmp.rules

protocol-telnet.rules

protocol-tftp.rules

protocol-voip.rules

pua-adware.rules

pua-other.rules

pua-p2p.rules

pua-toolbars.rules

rpc.rules

rservices.rules

scada.rules

scan.rules

server-apache.rules

server-iis.rules

server-mail.rules

server-mssql.rules

server-mysql.rules

server-oracle.rules

server-other.rules

server-samba.rules

server-webapp.rules

shellcode.rules

smtp.rules

snmp.rules

specific-threats.rules

spyware-put.rules

sql.rules

telnet.rules

tftp.rules

virus.rules

voip.rules

web-activex.rules

web-attacks.rules

web-cgi.rules

web-client.rules

web-coldfusion.rules

web-frontpage.rules

web-iis.rules

web-misc.rules

web-php.rules

x11.rules

F.2 Bleeding Snort rules:

http://www.bleedingsnort.com/downloads/bleeding.rules.tar.gz

bleeding-attack_response.rules

bleeding-dos.rules

bleeding-drop-BLOCK.rules

bleeding-drop.rules

bleeding-dshield-BLOCK.rules

bleeding-dshield.rules

bleeding-exploit.rules

bleeding-game.rules

bleeding-inappropriate.rules

bleeding-malware.rules

bleeding-p2p.rules

bleeding-policy.rules

bleeding-scan.rules

bleeding-virus.rules

bleeding-web.rules

bleeding.rules

F.3 Emerging Threats Snort rules:

http://www.emergingthreats.net/open-source/etopen-ruleset/

emerging-deleted.rules

emerging-dns.rules

emerging-dos.rules

emerging-drop.rules

emerging-dshield.rules

emerging-exploit.rules

emerging-ftp.rules

emerging-games.rules

emerging-icmp.rules

emerging-icmp_info.rules

emerging-imap.rules

emerging-inappropriate.rules

emerging-malware.rules

emerging-misc.rules

emerging-mobile_malware.rules

emerging-netbios.rules

emerging-p2p.rules

emerging-policy.rules

emerging-pop3.rules

emerging-rbn-malvertisers.rules

emerging-rbn.rules

emerging-rpc.rules

emerging-scada.rules

emerging-scan.rules

emerging-shellcode.rules

emerging-smtp.rules

emerging-snmp.rules

emerging-sql.rules

emerging-telnet.rules

emerging-tftp.rules

emerging-tor.rules

emerging-trojan.rules

emerging-user_agents.rules

emerging-virus.rules

emerging-voip.rules

emerging-web_client.rules

emerging-web_server.rules

emerging-web_specific_apps.rules

emerging-worm.rules

F.4 Vorant IRC Bot Snort rules:

http://www.vorant.com/files/irc-bot-snort-rules.txt}

irc-bot-snort-rules.rules