March 2020

Automated Extraction of Network Activity From Memory Resident Code

Austin Nicholas Sellers
Louisiana State University and Agricultural and Mechanical College

Automated Extraction of Network Activity From Memory Resident Code

Austin Nicholas Sellers

AUTOMATED EXTRACTION OF NETWORK ACTIVITY FROM MEMORY RESIDENT CODE

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

in

The Department of Computer Science and Engineering

by
Austin Nicholas Sellers
B.Sc., Louisiana State University and
Agricultural and Mechanical College, 2018
May 2020

Acknowledgments

I would like to thank Dr. Golden G. Richard III for guiding me as my advisor. His expertise and knowledge of all things memory forensics aided me throughout my research. I would like to thank Andrew Case for his help and guidance throughout my work. I would like to thank Dr. Konstantin Busch and Dr. Gerald Baumgartner for their contributions as my committee members. I would like to thank my friends and my colleagues in the cyber lab for their support in my research.

This work was supported by NSF grant "SaTC: CORE: Medium: Robust Memory Forensics Techniques for Userland Malware Analysis", Award # 1703683, $1,113,426.

Table of Contents

List of Figures

Abstract

Advancements in malware development, including the use of file-less and memory-only payloads, have led to a significant interest in the use of volatile memory analysis by digital forensics practitioners. Memory analysis can uncover a wealth of information not available via traditional analysis, such as the discovery of injected code, hooked APIs, and more. Unfortunately, the process of analyzing such malicious code is largely left to analysts who must manually reverse engineer the code to discover its intent. This task is not only slow and error-prone, but is also generally left only to senior-level analysts to perform, given that significant reverse engineering skills are required. This work focuses on the use of code emulation to automatically complete one of the most common tasks of malware analysis - discovering a malware sample's network activity. Our tool automatically discovers the locations where malware uses networking APIs, emulates the network operations, and records the parameters passed to those functions. Through the monitoring of such parameters, this work enables the automatic discovery of the IP addresses, domain names, and network ports utilized by malware to connect to remote command-and-control (C2) servers as well as accept incoming connections. This novel use of emulation applied to in-memory code provides significant benefits compared to traditional whole-system emulation, which requires a full executable to run and does not match the environment that malware executed during a live incident. In contrast, our approach can emulate any code in memory, including inside of shellcode buffers and memory-only libraries. The novel network API monitoring capabilities developed for this research project were written as an extension to HookTracer, which is an plugin for the Volatility memory analysis framework. HookTracer provides emulation of API hooks in memory, but does not target any specific network activity. The contribution of this work is the incorporation of network API monitoring into HookTracer, development of a test suite that ensures the parameter monitoring is correct, and the evaluation of the techniques we have developed against real-world malware.

Chapter 1. Introduction

1.1. Memory Forensics

The field of memory forensics (synonymous with the term memory analysis) is becoming increasingly relevant and necessary when performing modern digital forensics tasks. Memory forensics is the investigation of information stored in the volatile memory (RAM) of a computing device. Using memory forensics techniques, analysts can retrieve information that is often not available on the file system or in plaintext form on the network. This includes information like instructions of an executing process, encryption keys, network activity, etc. As malware of all varieties has become more sophisticated, however, the difficulty of malware analysis has likewise increased.

While memory analysis is generally performed in conjunction with traditional forms of digital forensics, such as analysis of file systems and network data, it many situations memory analysis is the only usable technique. This is particularly important with the increasing use of memory-only malware, which never writes any of the code that it executes to the file system. Some modern malware also has built-in security mechanisms that will prevent execution unless certain criteria have been met. One example of such a mechanism is malware that detects whether it is being run in a virtual machine environment or a debugger of some kind. If this is detected the malware may behave differently than it otherwise would. To combat such mechanisms the tools used by memory forensics practitioners need to be able to handle such cases. To prevent the spread of malware, analysis of malware obtained from memory samples is generally performed in an isolated environment, such as a virtual machine (VM). The isolation, however, can change the behavior of malware during its execution as it does not have outgoing internet access. The goal of this research was to locate and emulate networking code within malware residing in memory and "tricking" the malware's code into believing it has a connection and is executing on a real device.

1.2. Finding Network Connections

Analyzing memory samples can be a very time-consuming process, whether the analysis is conducted through command line based tools or by running scripts/plugins. These activities consume a significant amount of time by analysts, and increasing case work loads have brought the need for automated solutions. One such activity that memory forensics professionals frequently spend a large amount of time working on is searching for specific code segments that connect out to a domain/IP address associated with an intrusion. Gathering this network information allows for a wide range of tasks, such as developing host and network signatures, searching through logs for other victims, and blocking future requests to the malicious servers at the organization's firewall. Recovering the network configuration of malware is currently a mostly manual process that requires significant knowledge of operating system (OS) internals on part of the investigator. Such knowledge is not universal throughout the digital forensics community, and even for experienced investigators, the manual process is time-consuming, error-prone, and mentally taxing.

The presented research project streamlines this process by automatically locating calls of networking API functions and other surrounding code in a memory sample. After being located, these code segments are then emulated and the results are reported to the user along with the API parameters (domains/IP addresses/ports). The developed automation greatly reduces the effort and time spent by memory forensics practitioners in locating and parsing malware's network activity.

1.3. Contribution of This Thesis

This thesis presents an extension to HookTracer, which is a plugin to the Volatility memory analysis framework. HookTracer provides emulation of API hooks from a memory dump, but does not target any specific network activity. This work adds the ability to HookTracer to locate and emulate winsock networking API functions. The parameters sent to these functions are recorded and displayed to the user. To ensure that the added emulation features were operating correctly, a test suite was created. This test suite

contains a set of C programs that call the monitored Winsock functions with hard-coded parameters as well as a script that compares the hard-coded parameters to those captured by the emulator. To showcase the applicability of the research, tests were also run against malware samples that were discovered during real-world incident response.

1.4. Outline

Chapter 2 describes socket programming and traditional malware analysis techniques and explains our approach to hooking network APIs. Chapter 3 gives details on our test suite and test cases and how they were chosen and generated. Chapter 4 presents our results. Chapter 5 presents conclusions and outlines future work.

Chapter 2. Network APIs

2.1. Background and Literature Review

Modern socket APIs and socket programming can trace its roots to Berkeley Software Distributions (BSD) version 4.2. BSD was an evolution of UNIX, which was originally created within the research group of Bell Laboratories in 1969. After Version 7 was distributed within Bell Laboratories, the development was taken over by the UNIX Support Group (USG) within AT&T, which was a child company of Bell at the time. The first version released outside of the research community was System III in 1982. UNIX research continued at other institutions outside of AT&T in the years following, with the research at Berkeley being the most influential in the field of socket programming. Researchers at Berkeley received funding from the Defense Advanced Research Projects Agency (DARPA) to facilitate the creation of a version of UNIX for government use. This version of UNIX was built to support Transmission Control Protocol (TCP) and Internet Protocol (IP), protocols used for the DARPA internet (ARPANET) project [8]. Throughout their time developing BSD, Berkeley had several releases, including the release of 4.2 BSD in 1983, which introduced the idea of sockets and is the basis of most modern socket programming. Berkeley's last release was that of 4.4 BSD in June 1993 which formalized the POSIX standard that is used today [34].

2.1.1. Winsock API

In 1992, Microsoft released version 1.0 of their networking API, called the Windows Socket API (WSA) also known as Winsock [12]. Version 1.0 of this specification was not widely used however, until Microsoft released Winsock1.1 in January 1993. This version was also largely based on the Berkeley Sockets model laid out in the BSD releases. Microsoft did however create extensions to this model specific to Windows. These extensions facilitate multi-threading for Windows processes. The latest revision of the Winsock API

is Winsock2.2.2 (WSAPI22.DOC) [20] which was released in August 1997. Winsock2.2.2 has backwards compatibility with Winsock 1.1 and extends upon its functionality. Version 2.2.2 added many new features including support for asynchronous operations, completion routines, and support for other protocols like IPX and SPX, to name a few. Malware commonly uses the Winsock2 API for networking because it provides a simple and easy to use interface for UNIX compatible socket programming.

2.1.2. Uses of Winsock by Malware

There are higher level networking functions that malware can utilize to perform networking operations. However, when malware needs to have greater control over the sockets it uses for sending and receiving data, the API of choice is *ws2 32.dll* which is the Winsock2 DLL. Malware may use the functions provided by this DLL for simple command-and-control (C2) or data exfiltration, and such a low-level API gives malware significant control over the structure of its networking code. It is possible, but significantly more difficult, to extract this information using traditional analysis techniques as described below in Section 2.2. However, this is both more difficult and typically much more time consuming. Having a method to automatically extract information about the network connections that malware attempts to establish will prove to be invaluable to memory forensics practitioners.

2.2. Malware Analysis Techniques

When it comes to studying malware and its behavior, there are many methods of analysis. Some methods may not be applicable in a specific case depending on what information is available. One commonly used method is static analysis, which involves investigating the code and data associated with a malicious program without actually executing it. A second form of analysis is called dynamic analysis, which evolves the execution of a program under scrutiny, often in a controlled environment, to monitor its behavior. This includes behaviors such as reading and/or writing files, editing registry keys, and connecting to or attempting to connect to IP addresses/domains [6]. These aspects of malware analysis are further detailed in this section.

2.2.1. Static Analysis

Static analysis involves looking at the file structure of a malware sample, how it was compiled or packed, and digging deeper into the executable file's format and underlying structures. Most antivirus software also employs static analysis to determine if a file is malicious or not. They may search for file signatures, which are pieces of code from known malicious files, or they may use heuristics, which are patterns also known to be malicious [14]. A popular website used by researchers to gather results from many antivirus engines at once is VirusTotal [38]. VirusTotal runs uploaded files against over 70 antivirus scanners, returning discovered results to the user. VirusTotal and malware scanners are a good first step, however often further static analysis needs to be done on malware samples to fully understand them.

In the case of malware, further static analysis often comes in the form of analyzing assembly code, as it is not possible to retrieve the original language code in most cases. This assembly code is generated using a dissassembler, which takes an executable and renders human-readable assembly code. Static analysis of assembly code can be aided with tools such as the disassembler IDA Pro [13]. In some cases IDA can even convert into higher level languages, depending on the malware sample. Using tools like IDA Pro, the control flow of a program can also be followed and visual representations of control flow, such as control flow graphs, can be generated.

Once the assembly code is obtained however, analysis is not trivial. Reverse engineering (RE) malware by following its flow and understanding its intent is something that is normally reserved for more senior level analysts. Becoming proficient at reverse engineering typically requires significant knowledge and experience. Static analysis and reverse engineering however, only gets you so far, as the malware may have behavior that only occurs at run time. An example is when malware rewrites its own code as it runs, which may not be caught in static analysis. This may be discovered by reverse engineers, but is more easily discovered through execution. As stated by Moser et al. [23] in their review of some

limitations of static analysis, malware can use one of many code obfuscation methods, including opaque constants and block chaining or code transformation techniques, to hide control flow and data usage to avoid antivirus detection and other static analysis techniques. These obfuscation techniques make it more difficult for even experienced analysts to follow the flow of a malware sample and determine what it is trying to accomplish on the infected device.

2.2.2. Dynamic Analysis

When obfuscation techniques such as packing or encryption or run-time code modification are deployed by a malware sample, dynamic analysis is the method of choice for analysis. A collection of tools frequently used in dynamic analysis on Windows systems is sysinternals. The sysinternals suite was originally developed in 1996 by Bryce Cogswell and Mark Russinovich, as part of their company Winternals, which was later acquired by Microsoft [43] in 2006. Sysinternals has continued receiving updates to the included tools from Microsoft and the tools remain publicly available [27]. Two of the tools that are extremely useful in dynamic analysis are Process Explorer [25] and Process Monitor [24]. Process Explorer allows analysts to monitor the active processes and see which registry keys, files, DLLs or handles a specific process is currently accessing or has loaded. It is essentially a much more advanced task manager for Windows, which shows the current state of the machine and its processes. Process Monitor works similarly to Process Explorer but also keeps a log of process activities. It has the ability to log registry, file, thread, and network activities of all running processes to create a history that can be filtered and analyzed [32]. The Windows Sysinternals tools and many other dynamic analysis tools are largely manual and still require much analysis of the information that is generated.

Dynamic analysis can also be performed inside of a sandbox, which helps to automate the analysis process. One example of an open source sandboxed environment is Cuckoo Sandbox [7] which takes an executable as input and automatically provides the user with a report of the executable's behavior. This includes information on API calls made by the

executable, network traffic analysis, and analysis of the memory sample from the virtualized environment.

Chen et al. [5] demonstrated how Cuckoo could be used in the analysis of samples of the malware WannaCry. Using the output of Cuckoo Sandbox they were able to develop a set of features of WannaCry that could indicate if a device was infected with the virus. This analysis that would have been much more difficult to do using simple static analysis or reverse engineering. However, sandboxes are not always the most optimal tool to use for analysis. This is especially true if the analyst needs more fine-grained control over the environment the malware is running in.

2.2.3. Emulation

Emulation is another form of dynamic analysis which allows the analyst to have greater control over all aspects of the analysis. Emulators can be used to execute CPU instructions one by one and examine the result or instrument individual instructions to log results. Within an emulator, the analyst can control hardware-level aspects of the execution environment like CPU registers, the state of memory or RAM, etc. This is known as full system emulation and can even be used with an operating system running on top of the emulator. Kruegel [18] lays out some advantages of full system emulation over virtualization and other dynamic analysis techniques in his Black Hat 2014 presentation and paper. He explains how other techniques might miss certain behaviors of malware as it executes. With full system emulation there is greater visibility into the exact instructions executed by the CPU and the changes made to physical memory by a sample.

Traditional static and dynamic analysis is done when there is access to the executable of the malware in question. When this access is not available, other techniques must be used to evaluate the behavior of malware. Memory forensics techniques can be used in such a case to extract information from memory samples. This includes extracting information like instructions, function calls, and even entire executables from the RAM of an infected device. We note that not all modern malware uses traditional executable files, since shell-

code and memory-only malware are commonly encountered in the wild. Using emulation, these instructions and function calls can be taken from memory and can be run in an emulated environment without the need for full system or full executable emulation.

A common emulator that is used by analysts is the open-source emulator QEMU [1, 31]. QEMU provides interfaces for full-system as well as process/instruction emulation. However, full-system emulation can turn out to be very resource intensive, especially if there is an OS running on top of the full system being emulated. Furthermore, whole system emulation may not always be necessary to retrieve the necessary data for analysis. Since QEMU is open source, it has been extended upon and used in many malware analysis platforms to widen its functionality. The CPU emulator used in HookTracer on which our research is based is an extension of QEMU called Unicorn [28], introduced at Black Hat 2015. It provides a more lightweight and less resource intensive version of QEMU that provides bindings for most major languages.

Unicorn also allows emulation of arbitrary chunks of raw binary code and does not require a full system to be emulated. This is useful in our research as we only need to set up the parts of the environment necessary to execute the code segments we are looking at. If we needed to use full system emulation, the malware would be running in an environment that may have slight or major differences from the environment it was originally running in. Therefore, more parameters would need to be tweaked in the environment to facilitate successful execution. With Unicorn's ability to emulate arbitrary CPU instructions, we can tailor the environment to specific cases while also providing for maximum information extraction.

2.2.4. Analysis of Network Calls

Analysis of executables and the network connections they create can be a daunting task. To aid in this process there are a variety of tools/programs available today. However, in most research environments, malware is run in an isolated manner and does not have an active network connection. In such a case the network connections must be falsified

to trick the sample into performing network operations. An example of a tool with this capability is FakeNet-NG [17] created by the FireEye Labs Advanced Reverse Engineering (FLARE) team. It works by creating listeners and interfaces for common protocols used in networking. An executable is then run and the program returns the attempted network connection information. Our work takes a similar approach by returning fake data to the program to simulate a network connection.

There have been several other papers published regarding automated network analysis and connection emulation. Gorecki et al. [11] presents one method with TrumanBox, which is a tool that attempts to emulate the internet. They accomplish this by redirecting traffic to TrumanBox as a bridge between a client and the internet for dynamic malware analysis. During analysis, network information like IP addresses, ports, etc. can be extracted. The protocol being used can also be identified so traffic can be dispatched correctly, responding with static valid responses and logging all information gathered. This is done on the packet level and not on the function call level as is explored in our research.

A limitation of FakeNet-NG and the TrumanBox research is that they require an executable to run before they can return information about that executable and its behavior. Unfortunately, access to the executable associated with a malware infection may not always be feasible. For example, we may only have access to a memory sample from a device in which a piece of malware has been run. In such a case, FakeNet-NG, TrumanBox and similar tools may not be sufficient.

Static analysis in conjunction with dynamic analysis tools can be used to extract networking information from a malware sample. This is demonstrated by Pantazopoulos [29] in his analysis of Loki-Bot, which is a piece of malware that can perform keystroke logging and steal passwords and cryptowallets. It utilizes the Winsock framework for command and control and data exfiltration. In his research, reverse engineering of the assembly code is used in conjunction with WireShark to analyze the packets sent by the malware. However, this analysis required extensive knowledge of assembly language and reverse engineering

techniques, as well as a large amount of time, to walk through and analyze the flow of the malware sample. Analysis of malware samples such as the one discussed by Pantazopoulos can require a complex setup and a large amount of research time. Clearly, a lot of time and effort can be saved by automating the analysis and network activity extraction process using emulation.

2.3. HookTracer

Volatility [36] is a memory forensics framework written in Python that is used by many researchers to investigate memory samples. There are many plugins for Volatility written by the Volatility Foundation's [35] core developers and the Volatility community [37] that can be used for detailed analysis of memory. One such plugin useful in analysis of network activities is *netscan*, which uses pool tag scanning [19, Pages 129-135] to return a list of connections, including addresses, ports, and the owning processes that were active at the time the memory sample was taken. This plugin does not, however, always return information such as the owning process in cases where the executable has been terminated or if the connection has been closed, as shown in Figure 2.1. It also does not have the ability to find *where* in an executable or process the code for the connection was created and cannot extract the parameters or data sent by that process while the connection was active.

Offset(P)	Proto	Local Address	Foreign Address	State	Pid	Owner	Created
0xc40545ea5e80	TCPv6	:::49667	:::0	LISTENING	2004	svchost.exe	2019-11-26 17:01:27 UTC+0000
0xc4054879d2f0	TCPv4	0.0.0.0:49664	0.0.0.0:0	LISTENING	532	wininit.exe	2019-11-26 17:01:24 UTC+0000
0xc4054879ed30	TCPv4	0.0.0.0:49665	0.0.0.0:0	LISTENING	1164	svchost.exe	2019-11-26 17:01:26 UTC+0000
0xc4054ae0dbe0	TCPv4	192.168.79.129:49756	13.107.4.52:80	CLOSED	-1		-
0xc4054c16c160	UDPv4	127.0.0.1:56603	*:*		388	svchost.exe	2019-12-03 16:07:31 UTC+0000
0xc4054c16c550	UDPv6	::1:56601	*:*		388	svchost.exe	2019-12-03 16:07:31 UTC+0000
0xc4054c16c6a0	UDPv4	0.0.0.0:54391	*:*		5920	SkypeApp.exe	2019-11-26 17:40:28 UTC+0000
0xc4054c16c6a0	UDPv6	:::54391	*:*		5920	SkypeApp.exe	2019-11-26 17:40:28 UTC+0000
0xc4054c16c400	TCPv4	0.0.0.0:7680	0.0.0.0:0	LISTENING	5544	svchost.exe	2019-11-26 17:03:30 UTC+0000
0xc4054c16c400	TCPv6	:::7680	:::0	LISTENING	5544	svchost.exe	2019-11-26 17:03:30 UTC+0000
0xc405518114d0	TCPv4	127.0.0.1:50026	127.0.0.1:7171	CLOSE_WAIT	-1		-
0xc40561b06010	TCPv4	192.168.79.129:49997	40.69.220.46:443	CLOSED	-1		-
0xc405624e0590	TCPv4	0.0.0.0:49669	0.0.0.0:0	LISTENING	624	services.exe	2019-11-26 17:01:30 UTC+0000
0xc405624e0590	TCPv6	:::49669	:::0	LISTENING	624	services.exe	2019-11-26 17:01:30 UTC+0000
0xc405624e17f0	TCPv4	0.0.0.0:445	0.0.0.0:0	LISTENING	4	System	2019-11-26 17:01:30 UTC+0000
0xc405624e17f0	TCPv6	:::445	:::0	LISTENING	4	System	2019-11-26 17:01:30 UTC+0000
0xc4056a302450	TCPv4	192.168.79.129:50031	23.67.120.36:443	CLOSED	-1		-
0xc4057a4eec20	UDPv4	0.0.0.0:5355	*:*		1796	svchost.exe	2019-12-03 16:08:00 UTC+0000

Figure 2.1. Results of *netscan* Plugin

The research in this thesis presents an extension to HookTracer [4]. HookTracer is a plugin for Volatility that provides the ability to hook API calls extracted from a memory sample inside of an emulated address space. The mechanism of hooking allows our plugin to intercept function calls inside of an emulator and redirect them to versions of those API calls that are implemented internally while then providing the expected output back to the emulated program [22]. HookTracer also provides an interface for emulating functions and instructions using the emulation framework Unicorn [28].

Currently, HookTracer accomplishes this by taking as input the output of the apihooks Volatility plugin. With this input, HookTracer can obtain the addresses at which the APIs returned from those plugins begin. HookTracer then sets up the emulation environment by registering custom callbacks for important API functions and setting up the address space for emulation. After initializing the emulation environment and setting up the stack, HookTracer emulates from the addresses for the APIs input to the plugin from apihooks. It does this while mapping in pages from the memory sample into the emulator environment as they are accessed. A page is a block of memory that is of a fixed size. Analysis is then performed as the functions are emulated by mapping the basic blocks (code segments) of the functions into their specific memory regions. This analysis includes reporting on pages accessed by a specific API call, those pages' locations in memory, and their protection level (i.e., whether or not the page can be executed, read, or written to).

As HookTracer currently does not target specific networking APIs, our extension to HookTracer specifically addresses the Winsock2 API. It does this by automatically locating code segments in the memory sample where Winsock2 functions are called and emulating network calls along with other necessary supporting code on the fly. Currently, locating of API hooks within HookTracer leverages output from the apihooks plugin, beginning emulation using data from that plugin's output. However, when targeting Winsock functions, there are currently no plugins that can locate these API calls. Therefore, we implement a method of locating code segments that make Winsock API network calls. This facilitates normal

execution of the processes without the need for an active network connection. This can all be achieved in an environment without network connectivity using a memory sample from a device infected with malware that performs network operations.

2.4. Design and Implementation

Our extension to HookTracer is written in Python 2 and utilizes the Python binding provided by the Unicorn [28] emulation framework to integrate with the memory forensics framework Volatility [36]. As a prerequisite for running our plugin, a physical memory sample must be obtained from the device being analyzed. Acquisition of the memory sample can be performed using any standard method which generates a memory dump in a format supported by Volatility [42]. Once a memory sample of the device is obtained, our plugin can then be run to analyze the network calls made within the sample.

The following command line arguments are used to run our plugin:

- *Plugins directory* is the path to the folder in which our plugin is located. Volatility uses this argument to search for plugins of the name specified in this directory.
- *File* is the memory sample file that the plugin is to be run on.
- *Profile* identifies the profile to use on the memory sample. This profile tells Volatility which symbols, algorithms, and data structures to use in its analysis.
- *Plugin* is the name of the Volatility plugin to use on the memory sample. In our case the name of our plugin is hooktracer_net.

An example of the execution of our plugin is shown in Figure 2.2.

```
python vol.py --plugins="/austin/htnet/plugins" -f "/austin/htnet/image.vmem"
    --profile=Win10x64_15063 hooktracer_net
```

Figure 2.2. Plugin Arguments Example

The functions contained within the Winsock2.2 API have been well documented in Microsoft's official documentation [26]. Using this documentation, we were able to reconstruct the parameters and structures of the various functions within the Winsock2 API. During emulation within Unicorn, pointers and return values are setup by our plugin to be formatted as the program expects them. However, not all functions within the Winsock2 library need to be hooked by our plugin to successfully emulate and extract the parameters of the network function calls. The major functions that were hooked by our plugin are explained in Table 2.1. Most of the functions listed in Table 2.1 have variants specific to Winsock2 that include additional parameters such as conditions and additional data. These functions start with WSA followed by the function name to indicate that they are specific to the Windows implementation of sockets. For example, *WSAConnect*() allows for the sending of caller and callee data when the function is called in addition to the parameters of the *accept*() function. These Windows specific variants of the original BSD functions were also implemented as hooks in our plugin.

When the network API functions from the Winsock library are executed, there are auxiliary functions that Windows normally calls to handle certain actions internally. As these functions were not implemented within our emulated environment, hooks for these functions had to be implemented. This involved implementing functions like *_fileno* and *_lock_file* that would normally be provided by other libraries in Windows. These auxiliary functions were implemented to return values that would best allow emulation to continue.

To set up a connection with the Winsock2 API, the functions must be called in a specific order by the malware for a successful connection to be established. If the application is acting as a client connecting to a remote socket it must first call *WSAStartup*(), followed by *socket*(), *connect*(), and one of the *send*()/*sendto*() or *recv*()/*recvfrom*() function variants. If the malware is acting as a server which receives connections and listens on sockets it must again call *WSAStartup*(), followed by *socket*(), *bind*(), *listen*(), and finally one of the *accept*() variants before calling one of the *recv*()/*recvfrom*() or *send*()/*sendto*() functions.

Function	Function Description
WSAStartup	Loads Winsock DLL and allocates resources for network calls
WSACleanup	Unloads Winsock DLL and frees resources
WSAGetLastError	Returns the error code of the last networking call that failed
socket	Creates a socket of specified family, type, and protocol
connect	Establishes connection to a remote socket that is listening
WSAConnectByList	Establishes connection to one remote socket from a list
WSAConnectByName	Establishes connection to a remote host (service) and port
bind	Binds specified socket to a certain port
listen	Places specified socket in listening mode to accept connections
accept	Allows incoming connection on specified socket
send	Sends data to the remote socket
WSASendMsg	Sends data and control info to specified socket
WSASendDisconnect	Starts termination of connection with specified socket
sendto	Sends data to specified socket with destination address info
recv	Receives data from the remote socket
WSARecvDisconnect	Starts termination of connection with specified socket
recvfrom	Sends data to specified socket with source address info
shutdown	Disables sending and receiving on specified socket
closesocket	Closes an existing socket

Table 2.1. Major Winsock2 Functions Hooked by Plugin

A separate hook was created for each of the functions in the Winsock API because each of these functions expects different structures to be filled, return values, etc. Therefore, when one of the Winsock functions is called from within the malware sample, the code that is run is the custom hook for that function implemented within our plugin. With the return values satisfied, execution continues within the emulator and the parameters passed into that function are returned to the analyst.

The Winsock2 API keeps track of the state of sockets using socket descriptors [33]. These descriptors are the same as file descriptors in the UNIX implementation of sockets. However, with Winsock these socket descriptors are implemented as Windows handles. To handle socket descriptors as functions are called by the emulator, we implement an internal handle table to track the state of sockets. This internal socket table tracks the identifier (number) of the socket and if its state is new, connected, accepted, bound, listening, disconnected, or shutdown.

To locate each of the API functions, functions of the existing Volatility plugin *impscan* [40] were used. Impscan allows identifying where API calls are exported into the address space of the process that is being analyzed, as shown in Figure 2.3 It accomplishes this without having to look through the Import Address Table (IAT) of a Windows native portable executable (PE) and works even if a PE's IAT is paged out of memory.

```
IAT                Call               Module               Function
------------------ ------------------ -------------------- --------
0x00000000009ad000 0x0000000075b7a380 KERNEL32.DLL         TerminateProcess
0x00000000009ad004 0x0000000075b82b80 KERNEL32.DLL         IsDebuggerPresent
0x00000000009ad008 0x0000000075b80c30 KERNEL32.DLL         RaiseException
0x00000000009ad00c 0x0000000075b7efb0 KERNEL32.DLL         MultiByteToWideChar
...
0x00000000009ad058 0x0000000075b81930 KERNEL32.DLL         FreeLibrary
0x00000000009ad05c 0x0000000075b805a0 KERNEL32.DLL         GetProcAddress
0x00000000009ad0e4 0x0000000074895240 WS2_32.dll           WSACleanup
0x00000000009ad0ec 0x000000007488bfa0 WS2_32.dll           socket
0x00000000009ad0f0 0x00000000748908d0 WS2_32.dll           shutdown
0x00000000009ad0f4 0x0000000074885750 WS2_32.dll           send
0x00000000009ad104 0x000000007488f7a0 WS2_32.dll           getsockname
0x00000000009ad108 0x0000000074885650 WS2_32.dll           connect
0x00000000009ad10c 0x00000000748919a0 WS2_32.dll           WSAStartup
0x00000000009ad110 0x000000007488eac0 WS2_32.dll           closesocket
...
```

Figure 2.3. Impscan Results

This functionality allows our plugin to gather the location of the IAT for API calls for the processes being analyzed. From there, our plugin is able to search through memory to locate where these API function are actually called. The addresses of the exported API's IAT are obtained using Impscan's *enum_apis* function, which returns a list of all exported

APIs for the process being analyzed. To gather a list of the functions that were actually called, Impscan's *call_scan* function is called in conjunction with *_vicinity_scan* to search the process' memory regions for exports of APIs we are interested in that are called within the process, in this case Winsock2 functions. The full list of all exported functions, along with the current task and process address space, are passed into the *hook_analyzer* already provided in HookTracer. This sets up the basic infrastructure for emulating and hooking functions. This includes declaring hooks for functions/APIs we are monitoring, choosing the correct registers, offsets, and addresses depending on whether the code is 32-bit or 64-bit, and declaring functions used to read and write to and from the memory sample and the emulation environment.

To locate the addresses of the actual calls to Winsock APIs, memory regions are identified by examining process Virtual Address Descriptor (VAD) trees [9]. VAD trees are Windows data structure that describe pages allocated in memory to a process and how those pages relate to each other. When selecting VAD trees to traverse, Volatility's *vadinfo* [41] plugin is used to get information about the protection level of the VADs described by each node, as shown in Figure 2.4 This allows the plugin to target only those VAD entries that represent memory regions that are executable.

Using Volatility to walk the VAD trees the plugin can filter them to exclude VADs that map DLLs inside of the System32 or SysWOW64 directories. This saves significant time by skipping over legitimate Windows executables. The pages that contain the exported functions that were found by *impscan* are also excluded.

After this filtering has been applied, we are left with the VADs that are executable and are mapped to non-Windows DLLs and those that are executable but not backed by a file. At this point those executable VADs are searched for the opcodes FF 15 which is the Intel CALL instruction [16].

To locate a specific opcode in process memory, each filtered VAD is searched using the *zread* function in conjunction with the unpack function from struct (part of the Python

```
**********************************************************************
Pid:   6320
VAD node @ 0xffffc38beed26020 Start 0x00000000009c0000 End 0x00000000009cffff Tag Vad
Flags: Protection: 4
Protection: PAGE_READWRITE
Vad Type: VadNone
[snip]

VAD node @ 0xffffc38bee922b00 Start 0x0000000000990000 End 0x00000000009b1fff Tag Vad
Flags: Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @ffffc38beb610ce0 Segment ffffd40986c23870
NumberOfSectionReferences:          1 NumberOfPfnReferences:          17
NumberOfMappedViews:                1 NumberOfUserReferences:          2
Control Flags: File: 1, Image: 1
FileObject @ffffc38bef34e590, Name: \Device\Mup\;Z:000000000009bb39\vmware-host\Shared
                                    Folders\Shared\TcpClient.exe
First prototype PTE: ffffd40981403130 Last contiguous PTE: ffffd40981403238
Flags2: Inherit: 1, NoValidationNeeded: 1
...
```

Figure 2.4. *vadinfo* Results

standard library [30]). The *zread* function reads the specified number of bytes from process memory and unpack allows those bytes to be converted into an integer address.

After the CALL instruction is located, the plugin looks at the offset of the call and calculates the address of the function being called. Using the *unpack* function from the Python struct library it can unpack the binary data from memory directly after the CALL instruction as a little-endian integer value. Using this address, the plugin then checks to see if it matches the address of one of the Winsock API functions found by *impscan*.

If it is one of the functions that is being analyzed, the plugin traverses backwards in memory using a method similar to the method used to find the initial CALL opcodes. At this point the plugin is searching for the start of the function that contains the call to a Winsock API function. To locate the function start the plugin searches for the opcodes for push ebp (55) and mov ebp, esp (8B EC) for 32-bit executables or push rbp (55) and mov rbp, rsp (48 89 E5) for 64-bit executables [2, 3]. These instructions are known as the function prologue and are always called at the beginning of functions as a convention. This is to preserve the stack pointer before running the rest of the function, which is restored as the function returns. Emulation using Unicorn then starts at the location found for the function prologue

and stops a few instructions after the function call as shown in Figure 2.5. This way the entire executable does not need to be emulated to gather information about network calls. Through emulation the plugin is able to see the values of all of the parameters passed into the Winsock function that is being emulated and return this information to the user with the address of the calls for further analysis.

```
55                PUSH EBP                  void my_socket()
8b ec             MOV EBP, ESP              {              ← Emulation Begins
...                                             ...
ff 15 ec e0 81 00 CALL DWORD [0x81e0ec]         SendingSocket = socket(AF_INET,
                                                      SOCK_STREAM, IPPROTO_TCP);
3b f4             CMP ESI, ESP                  ...          ← Emulation Ends
...                                         }
```

Figure 2.5. Flow of Winsock Function Emulation

Once the location to start emulation at is found, the *analyze_address* function from HookTracer is called using that address as the start and ending a few instructions afterwards. Inside of the *analyze_address* function, functions are called that setup elements of the emulation environment, create hooks for code blocks, and overwrite our monitored functions in the memory space of the emulator. After this setup is complete, emulation actually begins at the specified address while mapping in missing pages from the memory sample as needed and redirecting API calls to our internal hooks and reporting on the results.

Chapter 3. Test Cases

3.1. Functionality Testing Suite

The process of creating test cases for our research involved writing corresponding client and server programs in C that call the functions we are hooking in the plugin. These test programs include calls to all of the Winsock2 API hooks, allowing us to individually test our function hooks with known input to those functions. These programs were compiled using Visual Studio [21] into 32-bit and 64-bit executables. VMWare Workstation Pro [39] was used to create and run a virtual machine with a clean installation of Windows 10. The compiled test executables were then run within this VM and the VM was suspended. Once suspended, the memory sample (*.vmem*) and suspended state file (*.vmss*) were gathered for analysis as input into the plugin.

To get complete coverage of the functions in the Winsock2 library that we are hooking we created several different test programs. One set of client/server programs was implemented with the BSD compatible methods like *socket*() and *connect*() and another set was implemented to use the more advanced Windows specific versions of these functions like *WSASocket*() and *WSAConnect*().

Within each of these test programs information about the parameters passed into each of our functions of interest were written to an output file. The information written includes data on the parameters like IP addresses and port numbers passed to the functions along with the process id of the executable. This information is used by our testing suite to verify the output of our known test cases matches the output of our plugin after emulating the located functions using Unicorn.

Our test suite takes as input the information output about API calls and their parameters from our test client/server programs, as well as, the output obtained once our plugin is run and emulation is complete. To verify the results with the known good data obtained

from our client/server programs, the test suite parses the output from both the programs and plugin to get them in a usable format. It then compares the values of parameters obtained from the programs and plugin to each other and returns a report of any differences between them. Using this test suite we now have a simple and easy way for verifying the workflow of our plugin. This is important to ensure that the Winsock functions are hooked correctly within our plugin and the parameters are able to be obtained through emulation and reported back to the user.

3.2. Malware Testing

For a more real world testing scenario and to test the efficacy of our HookTracer extension in extracting network information, we infected VMs with known malware samples obtained from VirusTotal. Research has previously been done on this malware and therefore the domains/IP addresses they connect to were known for validation of our API hooks. One of the malware samples is suspected to be a part of a malware campaign focusing on U.S. engineering and maritime targets by Chinese APT group TEMP.Periscope AKA "Leviathan" and has the hash 7ba05abdf8f0323aa30c3d52e22df951eb5b67a2620014336eab7907b0a5cedf. During the course of the malware campaign against U.S. engineering and maritime targets for cyber-espionage, many different pieces of malware were used, one of which was a backdoor called BADFLICK, used to modify the target's filesystem and generate a reverse shell to communicate with its C2 server [10]. Spread of this malware is done through an infected Word document which uses Microsoft Equation Editor and process hollowing to infect the device and create a backdoor connection [15]. To accomplish this communication, the Winsock network API functions from *ws2 32.dll* are used for C2 from which our plugin is able to recover information about these calls.

Chapter 4. Results

4.1. Test Program Results

The first output that is gathered is that which was output by the test client/server programs that were run inside of the VMs we are analyzing. This information is in the form of a comma separated list that contains the process ID (PID), name of function called, and the parameters passed to that Winsock function within the C executables as shown in Figure 4.1.

```
1520 socket, AF_INET, SOCK_STREAM, IPPROTO_TCP
1520 connect, localhost, 127.0.0.1, 7171
1520 getsockname, AF_INET
1520 inet_ntoa, localhost, 127.0.0.1
1520 send, This is a test string from sender
1004 socket, AF_INET, SOCK_STREAM, IPPROTO_TCP
1004 bind, 7171
```

Figure 4.1. Snippet of Test Program Log Output

Using the *.vmem* memory images obtained from a VM in which our client/server pairs were run, Volatility is run specifying the memory image and our plugin as input. After emulation is complete using Unicorn, the resulting output consists of all functions that were called over the course of emulation. The plugin output includes the PID of the process that ran the malware in question, the name of the executable, the function name of the API that is called, the address of the function being called and any parameters that were passed to the function. During emulation our plugin intercepts calls to Winsock API functions and redirect them to hooks implemented internally that extract parameters from those calls. This also gives us control over what to return to the executable being emulated. As we want emulation to proceed as far as possible, the values returned always indicate that the call was successful. A snippet of the output of the plugin is shown in Figure 4.2, including the parameters extracted from the Winsock functions.

```
1004 TcpServer.exe   WSAStartup    0x773e19a0L
1004 TcpServer.exe   socket        0x773dbfa0L   Family: AF_INET - Type: SOCK_STREAM - Protocol: IPPROTO_TCP
1004 TcpServer.exe   bind          0x773dcda0L   Port: 7171
1004 TcpServer.exe   listen        0x773d54b0L
...
1520 TcpClient.exe   connect       0x773d5650L   Domain: localhost - IPv4 Addr: 127.0.0.1 - Port: 7171
1520 TcpClient.exe   getsockname   0x773df7a0L   Family: AF_INET (2)
1520 TcpClient.exe   inet_ntoa     0x773e6eb0L   Domain: localhost  - IPv4 Addr: 127.0.0.1
1520 TcpClient.exe   send          0x773d5750L   This is a test string from sender
1520 TcpClient.exe   closesocket   0x773deac0L
```

Figure 4.2. Snippet of HookTracer Network API Results

The output of the plugin, in addition to the output from the test programs themselves, can then be used as inputs to our testing suite. The testing suite parses each of the outputs extracting information on the functions called and the parameters used in both execution of the programs and emulation through the plugin. Since the plugin output also includes additional functions that are not part of the Winsock library, the results are filtered to target only Winsock functions. These elements are then put into a list that contains the PIDs and their associated function calls with included parameters.

With one list for the known-good program output and one list for the plugin output, these lists are compared and the resulting output that is generated indicates whether or not the outputs match and what the differences are, as shown in Figure 4.3.

```
==== PROGRAM OUTPUT ====
  PID: 1520
    ('connect', 'localhost', '127.0.0.1', 7171)
    ('getsockname', 'AF_INET')
    ('inet_ntoa', 'localhost', '127.0.0.1')
    ('send', 'This is a test string from sender')
    ('socket', 'AF_INET', 'SOCK_STREAM', 'IPPROTO_TCP')
  PID: 1004
    ('bind', 7171)
    ('socket', 'AF_INET', 'SOCK_STREAM', 'IPPROTO_TCP')
==== PLUGIN OUTPUT====
  PID: 1520
    ('connect', 'localhost', '127.0.0.1', 7171)
    ('getsockname', 'AF_INET')
    ('inet_ntoa', 'localhost', '127.0.0.1')
    ('send', 'This is a test string from sender')
    ('socket', 'AF_INET', 'SOCK_STREAM', 'IPPROTO_TCP')
  PID: 1004
    ('bind', 7171)
    ('socket', 'AF_INET', 'SOCK_STREAM', 'IPPROTO_TCP')
==== DIFF ====
  OUTPUTS MATCH
```

Figure 4.3. Results from Testing Suite

The results from the test suite allow us to validate that each of the Winsock functions is correctly hooked within the plugin. This allows us to show that even though the functions are being hooked, normal execution in the emulator still occurs, creating the same results as if the programs were running on a normal OS or in this case inside of a VM.

4.2. Malware Sample Results

From the malware sample used by the APT group TEMP.Periscope/"Leviathan", we were able to extract information about the IP address to which connection was attempted and the type of socket used for C2 by the malware. These results are shown in our plugin output in Figure 4.4.

```
2092 7ba05abdf8f032  socket   0x76813eb8  Family: AF_INET - Type: SOCK_STREAM - Protocol: IPPROTO_TCP
2092 7ba05abdf8f032  WSAIoctl 0x76812fe7
...
2092 7ba05abdf8f032  ntohs    0x76812d8b
2092 7ba05abdf8f032  connect  0x76816bdd  Domain: 103.243.175.181 - IPv4 Addr: 103.243.175.181 - Port: 80
```

Figure 4.4. BADFLICK - HookTracer Network API Results

The BADFLICK backdoor attempts connection to the IP address 103.243.175.181 using port 80 over a TCP socket. This IP address is known to be the address of the BADFLICK C2 server, as shown in the analysis by IBM X-Force IRIS [15]. In this case analysis had previously been done on the malware, which we used to verify our results. However, this demonstrates the efficacy HookTracer extension in extracting network connection information through emulation.

Chapter 5. Conclusions and Future Work

5.1. Conclusions

The extraction of network activity associated with malware from memory resident code has always been a mostly manual and error-prone process. This process has traditionally involved extensive reverse engineering and static code analysis or the emulation of an entire executable from memory to locate information on network activity. When investigators only have access to a memory dump of a machine, traditional methods of analysis as described in Section 2.2 may not provide enough coverage and may be too slow or error-prone to be viable.

In our research, we show that emulation provides a robust method of extracting network information from memory resident code. Having a tool that can quickly extract networking information and report this to the user in a easy to digest and understand manner is vital. The results obtained from our enhancements to the HookTracer plugin show that the task of locating and extracting the network activity of malware from volatile memory no longer has to be a slow and manual process.

The plugin is able to automatically identify the locations in a memory sample for a process where Winsock network APIs were called by locating the start of the enclosing function and through emulation, extracting the parameters passed to those functions. As a result domain names, IP addresses, ports, and data can be reported to investigators for further analysis.

In this paper we discussed our method of validating our API hooks using our test suite as described in Section 3.1. This allows us to compare our plugin to known good data and validate results. This is important not just for the current state of the plugin, but also for creating an easy and robust way to validate hooks for functions that will be added in the future using custom testing programs.

We also were able to analyze malware samples using our plugin, showing that it is possible to extract networking information from real-world malware samples. This demonstrates the effectiveness of our plugin and its usefulness in memory forensics and incident response.

5.2. Future Work

To provide even more coverage of malware's use of networking APIs, the plugin could be extended to record each malware function that calls a networking API and then scan for calls to that function from inside the malware. These outer functions could then be the starting point of emulation. While many malware samples rely on global variables to use as parameters for API calls, there are also samples that use parameters from other internal functions to then pass to the networking APIs. By recursively scanning for the functions that call these functions of the malware, HookTracer would have better API tracing coverage as well as be able to recover the specific parameters used by malware coded in this manner.

A secondary plan of future work involves creating a more streamlined testing process for analysing malware samples. While the known-good test suite had to be hand-developed for the values to be hard-coded, such a labor intensive process is not needed when analyzing a variety of malware samples. For example, many industry researchers will publish indicators of compromise (IOCs) related to malware they analyze that describe all of the malware's activity. This can include artifacts, such as files created, registry keys modified, and network activity. By parsing the network-related artifacts from IOC databases and comparing them to the emulator's results, the validity and correctness of the emulation can be immediately determined. Other sources of known-verified network activity can include reports from sandbox environments, PCAP or NetFlow captures, and other available report formats, such as STIX.

Another expansion of the project would be expanding our plugin to extract parameters from more network APIs. While this work was largely focused on APIs from Winsock, there are other subsystems, such as DNS and WinInet, whose APIs provide functionality

that malware can use to perform a wide range of network activity. Also, although our plugin currently only supports Windows-based malware, the knowledge gained during this research would allow for direct transition of the work to design of a plugin capable of supporting the networking APIs and frameworks for MacOS and Linux. This would further increase our coverage of the malware landscape.

References

[1] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[2] A. Bloomfield. (2017) The 32 bit x86 c calling convention. [Online]. Available: https://aaronbloomfield.github.io/pdr/book/x86-32bit-ccc-chapter.pdf

[3] A. Bloomfield. (2017) The 64 bit x86 c calling convention. [Online]. Available: https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf

[4] A. Case, M. M. Jalalzai, M. Firoz-Ul-Amin, R. D. Maggio, A. Ali-Gombe, M. Sun, and G. G. Richard, "Hooktracer: A system for automated and accessible api hooks analysis," *Digital Investigation*, vol. 29, pp. S104 – S112, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287619301604

[5] Q. Chen and R. Bridges, "Automated behavioral analysis of malware: A case study of wannacry ransomware," 12 2017, pp. 454 – 460.

[6] Comodo Security Solutions, Inc. (2020) What is malware analysis — different tools for malware analysis. [Online]. Available: https://blog.comodo.com/malware/different-techniques-for-malware-analysis

[7] Cuckoo Sandbox. (2020) Cuckoo sandbox - automated malware analysis. [Online]. Available: https://cuckoosandbox.org/

[8] DARPA. (2020) Foundation of tcp/ip. [Online]. Available: https://www.darpa.mil/about-us/timeline/tcp-ip

[9] B. Dolan-Gavitt, "The vad tree: A process-eye view of physical memory," *Digital Investigation*, vol. 4, pp. 62 – 64, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287607000503

[10] FireEye. (2018, Mar) Suspected chinese cyber espionage group (temp.periscope) targeting u.s. engineering and maritime industries. [Online]. Available: https://www.fireeye.com/blog/threat-research/2018/03/suspected-chinese-espionage-group-targeting-maritime-and-engineering-industries.html

[11] C. Gorecki, F. Freiling, M. Kührer, and T. Holz, "Trumanbox: Improving dynamic malware analysis by emulating the internet," vol. 6976, 10 2011, pp. 208–222.

[12] M. Hall, M. Towfiq, G. Arnold, D. Treadwell, and H. Sanders. (1992) An open interface for network programming under microsoft windows. [Online]. Available: http://www.sockets.com/winsock.htm

[13] Hex Rays. (2020) Ida pro - hex rays. [Online]. Available: https://www.hex-rays.com/products/ida/

[14] A. Honig and M. Sikorski, *Practical Malware Analysis*. No Starch Press, February 2012.

[15] IBM X-Force IRIS. (2019, Jul) Badflick analysis report. [Online]. Available: https://exchange.xforce.ibmcloud.com/malware-analysis/4005cb35be10d65ee4b5773907f736c7

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Oct 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[17] P. Kacherginsky. (2016, August) Fakenet-ng: Next generation dynamic network analysis tool. [Online]. Available: https://www.fireeye.com/blog/threat-research/2016/08/fakenet-ng_next_gen.html

[18] C. Kruegel. (2014) Full system emulation: Achieving successful automated dynamic analysis of evasive malware. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Kruegel-Full-System-Emulation-Achieving-Successful-Automated-Dynamic-Analysis-Of-Evasive-Malware-WP.pdf

[19] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st ed. Wiley Publishing, 2014.

[20] Microsoft, *An Interface for Transparent Network Programming Under Microsoft WindowsTM*, 1997. [Online]. Available: http://www.comp.hkbu.edu.hk/~comp2330/onlineResources/WSAPI22.doc

[21] Microsoft. (2020) Visual studio ide, code editor, azure devops, & app center - visual studio. [Online]. Available: https://visualstudio.microsoft.com/

[22] MITRE ATT&CK™. (2019) Hooking - enterprise — mitre att&ck™. [Online]. Available: https://attack.mitre.org/techniques/T1179/

[23] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," 01 2008, pp. 421 – 430.

[24] MSDN. (2020) Process explorer - windows sysinternals. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/

[25] MSDN. (2020) Process monitor - windows sysinternals. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/

[26] MSDN. (2020) Windows sockets 2. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/winsock/windows-sockets-start-page-2

[27] MSDN. (2020) Windows sysinternals. [Online]. Available: https://docs.microsoft.com/en-us/sysinternals/

[28] A. Q. Nguyen and H. V. Dang, "Unicorn: Next generation cpu emulator framework," in *Proceedings of the 2015 Blackhat USA conference*, 2015. [Online]. Available: https://www.blackhat.com/docs/us-15/materials/us-15-Nguyen-Unicorn-Next-Generation-CPU-Emulator-Framework.pdf

[29] R. Pantazopoulos, "Loki-bot: Information stealer, keylogger, & more!" in *SANS Institute Information Security Reading Room.* SANS Institute, 2017, pp. 97 – 103. [Online]. Available: https://www.sans.org/reading-room/whitepapers/malicious/loki-bot-information-stealer-keylogger-more-37850

[30] Python Software Foundation. (2020) struct — interpret strings as packed binary data. [Online]. Available: https://docs.python.org/2/library/struct.html

[31] QEMU. (2020) Qemu. [Online]. Available: https://www.qemu.org/

[32] M. Russinovich and A. Margosis, "Windows sysinternals administrator's reference," 2011.

[33] Science Direct. (2020) Socket descriptor - an overview — sciencedirect topics. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/socket-descriptor

[34] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. John Wiley & Sons, Inc., December 2012.

[35] The Volatility Foundation. (2020) The volatility foundation - open source memory forensics. [Online]. Available: https://www.volatilityfoundation.org/

[36] The Volatility Foundation. (2020) Volatility framework - volatile memory extraction utility framework. [Online]. Available: https://github.com/volatilityfoundation/volatility

[37] The Volatility Foundation. (2020) Volatility plugins developed and maintained by the community. [Online]. Available: https://github.com/volatilityfoundation/community

[38] VirusTotal. (2020) Virustotal. [Online]. Available: https://www.virustotal.com

[39] VMWare. (2020) Vmware workstation pro. [Online]. Available: https://www.vmware.com/products/workstation-pro.html

[40] Volatility Foundation. (2017) Command reference mal - impscan. [Online]. Available: https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#impscan

[41] Volatility Foundation. (2019) Command reference - vadinfo. [Online]. Available: https://github.com/volatilityfoundation/volatility/wiki/Command-Reference#vadinfo

[42] Volatility Foundation. (2020) Faq—volatilityfoundation. [Online]. Available: https://www.volatilityfoundation.org/faq

[43] Winternals Software LP, "Microsoft acquires winternals software," *Company Press Releases*, 7 2006. [Online]. Available: https://web.archive.org/web/20070314051129/http://www.winternals.com/Company/PressRelease92.aspx

Vita

Austin Sellers was born in Baton Rouge, Louisiana. He graduated in May 2018 from Louisiana State University with his B.Sc. in Computer Science - Software Engineering. In August 2018, he started his master's studies in Computer Science at the Division of Computer Science and Engineering at Louisiana State University. He anticipates graduating with his M.Sc. in May 2020.