Improving Memory Forensics Through Emulation and Program Analysis

Ryan Dominick Maggio
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_dissertations

Part of the Information Security Commons

Recommended Citation
https://repository.lsu.edu/gradschool_dissertations/5528

This Dissertation is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Doctoral Dissertations by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.
IMPROVING MEMORY FORENSICS
THROUGH EMULATION AND PROGRAM ANALYSIS

A Dissertation

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
Doctor of Philosophy

in

The Division of Computer Science and Engineering

by

Ryan Dominick Maggio
B.S., Louisiana State University, 2016
May 2021
To my parents, Frank and Karen, for helping me become the person I am today.
Acknowledgments

First and foremost, I would like to thank Dr. Golden Richard, III for all manner of help in pursuing this degree. Your support as an advisor, a mentor, and a friend has been invaluable over the past five years. This was a daunting undertaking, but you helped make it not only manageable, but fun. I would also like to thanks Andrew Case and Dr. Aisha Ali-Gombe for the effort they put towards pushing forward research for the lab in general, and for me personally. The work done with this lab would not be possible without your contributions. Similarly, I would like to thank all of my coworkers in the lab, past and present, for their support, guidance, and friendship over the years. Additional thanks to my doctoral committee, Dr. Feng Chen, Dr. Mingxuan Sun, and Dr. Ye-Sho Chen for your valuable feedback on my work, and your time and patience throughout this process. Special thanks to Dr. Costas Busch for your help and guidance during your time at LSU, and your efforts on my doctoral committee. I would further like to thank Dr. Jianhua Chen for her support in both my undergraduate and graduate careers, for pushing me to continue pursuing computer science, and for her advocacy for my admittance into graduate school.

For all of his efforts as Department Chair, I would like to extend special thanks to Dr. Bijaya Karki. Innumerable thanks to my parents, Frank and Karen, my brother, Michael, and my grandmother, Lorraine. Your constant love, encouragement, and guidance carried me through this process many times. Thanks to my son, Xavier, for too much to list, but especially for being the best motivation to push forward I could ever have. Thanks to my partner, Rachel, for your love, and for keeping me centered. Finally, thanks to all of my friends who have stuck by me over the years. You’ve all been there for me through both the hardest and the most rewarding parts of my life, and I wouldn’t be here today without you. The list is too long to present in its entirety, but special thanks to Alex, Clayton, Alice, Michelle, John, and Laila. You are all family to me.
# Table of Contents

Acknowledgements ................................................. iv

List of Tables ......................................................... vi

List of Figures ......................................................... vii

Abstract ............................................................... ix

Chapter 1: Introduction ................................................ 1
  1.1 Memory Forensics ................................................. 1
  1.2 Hooks ............................................................ 4
  1.3 Code Injection ................................................... 7
  1.4 Concrete and Symbolic Execution ............................... 9

Chapter 2: HookTracer ................................................. 13
  2.1 Introduction ..................................................... 13
  2.2 hooktracer ....................................................... 14
  2.3 hooktracer_messagehooks ...................................... 23
  2.4 Iodine ............................................................ 39
  2.5 Related Work .................................................... 54

Chapter 3: Seance ....................................................... 57
  3.1 On Reading Bones ................................................ 59
  3.2 How to Commune With Spirits ................................. 62
  3.3 Conversations with Dead People ............................. 73
  3.4 Related Work .................................................... 85

Chapter 4: Conclusion ................................................ 90
  4.1 Here Be Dragons ................................................. 91
  4.2 Closing Remarks ............................................... 92

Appendix A: Publishing Agreements ................................... 93

References ............................................................ 100

Vita ................................................................. 110
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Number of API Hooks by Operating System</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Obj-C data structures targeted</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>Part 1 of a long table of Objective C functions analysed</td>
<td>75</td>
</tr>
<tr>
<td>3.3</td>
<td>Part 2 of a long table of Objective C functions analysed</td>
<td>76</td>
</tr>
<tr>
<td>3.4</td>
<td>TCP_ENDPOINT members and referencing functions</td>
<td>82</td>
</tr>
<tr>
<td>3.5</td>
<td>Windows 10 versions and build numbers</td>
<td>83</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 SetWindowsHookEx prototype ........................................ 6  
1.2 Example code .......................................................... 10  
2.1 Inline hook ............................................................. 15  
2.2 *hooktracer* on Coreflood .......................................... 19  
2.3 *hooktracer* on TDSS .................................................. 19  
2.4 *hooktracer* on AVG .................................................... 20  
2.5 Visual C++ runtime API hook ....................................... 21  
2.6 OneDrive API hook .................................................... 21  
2.7 *hooktracer* output for Zeus ....................................... 22  
2.8 *hooktracer* grouping Zeus hooks ................................ 23  
2.9 Global message hook ................................................ 24  
2.10 Local message hook .................................................. 25  
2.11 KeyboardProc prototype .......................................... 29  
2.12 Gozi snippet .......................................................... 30  
2.13 *hooktracer_messagehooks* against Turla ...................... 33  
2.14 Turla Trace Record .................................................. 34  
2.15 *hooktracer_messagehooks* against Loki Bot .................. 36  
2.16 *hooktracer_messagehooks* against Gozi ....................... 37  
2.17 *hooktracer_messagehooks* against Donot Team’s Keylogger. 38  
2.18 Malfind output from Stuxnet infected image .................. 40  
2.19 iodine output from Zeus infected image ......................... 45
Abstract
Memory forensics is an important tool in the hands of investigators. However, determining if a computer is infected with malicious software is time consuming, even for experts. Tasks that require manual reverse engineering of code or data structures create a significant bottleneck in the investigative workflow. Through the application of emulation software and symbolic execution, these strains have been greatly lessened, allowing for faster and more thorough investigation. Furthermore, these efforts have reduced the barrier for forensic investigation, so that reasonable conclusions can be drawn even by non-expert investigators. While previously Volatility had allowed for the detection of malicious hooks and injected code with an insurmountably high false positive rate, the techniques presented in the work have allowed for a much lower false positive rate automatically, and yield more detailed information when manual analysis is required. The second contribution of this work is to improve the reliability of memory forensic tools. As it currently stands, if some component of the operating system or language runtime has been updated, the task of verifying that these changes do not affect the correctness of investigative tools involves a large reverse engineering effort, and significant domain knowledge, on the part of whoever maintains the tool. Through modifications of the techniques used in the hook analysis, this burden can be lessened or eliminated by comparing the last known functionality to the new functionality. This allows the tool to be updated quickly and effectively, so that investigations can proceed without issue.
Chapter One. Introduction

1.1 Memory Forensics

Memory forensics [75] is the process of examining the contents of a computer’s Random Access Memory (RAM), sometimes called ‘main memory’ or simply ‘memory,’ to discover some information about the activities on that computer at the time the memory dump was taken. Reasons that an investigator might want this information vary, but commonly it is to either find evidence of a crime committed by the user, or to discover the presence of malware on the computer. Memory forensics is a subset of the field of digital forensics, and has seen an increase in use in recent years as a response to such things as memory-only malware infecting systems, such as many attacks using Powershell Empire [107], or criminals using cloud-based communications and file storage, which leave no evidence on disk. By themselves, disk or network forensics have never given a complete picture, but developments such as these have made it so that by ignoring memory forensics you may well lose all artifacts relevant to an investigation and not even know it. However, the tools used in memory forensics are still fairly young, and have much room for improvement. The focus of this work is to push the boundaries of what can be accomplished by tools such as Volatility [128] so that investigations can be completed more quickly, accurately, and in greater depth.

1.1.1 Volatility

The Volatility memory forensics framework, mentioned in the previous section, is the most widely used memory forensics framework. Due to its ease of use, open source nature, and plugin-based ecosystem, it is the only memory forensics tool that will be discussed in this work, and is the only one used for the research presented here. As such, it merits a brief introduction. Volatility is used to analyze memory images, but does not provide functionality to collect them. As mentioned, Volatility is an open source tool hosted on Github, and provides functionality based on which plugins are installed. Plugins provided by core devel-
opers make up most of capabilities provided by Volatility, but many user-made plugins are also included on the Github repository. Additionally, users can develop their own plugins for personal use, share them with the community, or even work with the core developers to get the plugins added to the standard Volatility installation. Currently, Volatility can analyze memory dumps of all popular versions of the Windows and OSX operating systems, as well as most popular Linux distributions, but due to its plugin-based nature, not all of the functionality available to one operating system version will be available to another. In the scope of this work, most of the focus will be on Windows analysis unless otherwise specified. This choice was made due to the relative popularity of Windows for both consumer and enterprise use as well as the number of compatible and notable malware that target Windows systems.

1.1.2 As Program Analysis

When discussing program analysis methods, people generally talk about two general strategies or methodologies: static analysis and dynamic analysis. Static analysis [109, 27] covers things such as reading through a program’s source code, or perhaps the disassembly of some program’s binary. More generally, static analysis describes methods of analysis that aim to study a program without actually executing code. Dynamic analysis includes taint analysis [39], sandboxing [138], or otherwise observing the behavior of some code by running it. There are relative strengths and weaknesses to both approaches. Static analysis can be significantly more time consuming, and code obfuscation or encryption can render these methods effectively impossible, where something as simple as a three hour sleep at the beginning of some process can be enough to convince a dynamic analyst that the code did not run correctly. On the other hand, static analysis gives a very complete picture of the structure of a program, and does not risk a malware infection on the analyst’s own machine, while dynamic analysis can quickly reveal the most relevant behaviors of some code, no matter how unreadable it was made to be. As will be covered later, while static and dynamic are sometimes thought of as "the two" forms of analysis, this is not strictly true. It is, however,
a useful simplification in many situations. When looked at this way, analysis of a memory image occupies kind of a fused space of these broad domains. In many cases, the raw code of some program is, indeed, present in memory, and can be understood as a static piece of code. Simultaneously, because the code was actually being run, and because the view that memory forensics gives includes operating system structures and behaviors, an investigator can get information about how the program behaves when run. For instance, network connections initialized by the process, Dynamically Linked Libraries (DLLs or dlls) loaded, and other information that might be hard, or impossible, to deduce from simply reading code is readily available. Some malware can detect the presence of virtualization software, debuggers, and other staples of various dynamic analysis toolkits [42, 25]. When analyzed inside of a memory image, however, one can observe the "true" behavior of the malware since it has already been executed in an environment without instrumentation. Furthermore, since it is no longer running it cannot actively thwart attempts at observation. This is not to say that the analysis of a memory image solves all problems facing other analysis methods, or retains their relative strengths. Outside the case of virtual machines, acquiring a memory image is a non-trivial and error-prone process [9, 129, 52, 50]. When trying to analyze the image of a virtual machine, however, you still have to contend with the same anti-analysis techniques that make dynamic analysis difficult. Additionally, the code present in memory is still a binary at best, and in some cases may not even be entirely present in memory. In summary, program analysis through analysis of a memory image can provide additional insights or benefits that other techniques cannot, but it is not a catch-all. However, in the case of forensic investigations there may not be an additional option, as a program binary may not be available on non-volatile storage.
1.2 Hooks

A large amount of this research involves the analysis of hooks installed on a system. Broadly, the research is focused on two types of hooks, function hooks (or API hooks) and event hooks. Function hooks involve replacing one function, say \(f_{\text{orig}}\), with another, say \(f_{\text{hooked}}\), so that when \(f_{\text{orig}}\) is called, \(f_{\text{hooked}}\) is executed instead. While this may appear a solely malicious process, legitimate uses include hooking your own code for backwards compatibility or system monitoring, for example by antivirus software. On the other hand, malware may install API hooks in order to maliciously alter the behavior of some program or, more commonly, to spy on the user. In this case, \(f_{\text{hooked}}\) eventually returns control to \(f_{\text{orig}}\), but since it executes first, and with the same parameters, it can get information about user behavior. For example, by hooking networking functions, malware might be able to determine what websites a user is visiting, or if it takes a more active role, it could redirect the user to a malicious website. Event hooks, on the other hand, make use of hooking functionality provided by the operating system itself to install the hooks. Instead of overwriting code or data somewhere, installing an event hook means adding the hook to a queue of other hooks on the system, all listening for a specific type of event. For instance, one could install hooks that execute whenever a key is pressed, whenever a USB device is inserted, or a host of other things. Once again there are both legitimate and malicious uses for this type of hook, which is why the operating system provides functionality to install them. For example, any program that relies on hot-keys might install a hook that fires on key presses, while a keylogger (malicious or otherwise) might install a similar hook to record that key press. The following sections will provide a brief overview of the different types of hooks this research focuses on.
1.2.1 Inline Hooks

Inline or trampoline hooks are a simple type of API hook, where some part of the original function, usually at the very beginning is overwritten with instructions that alter the control flow and redirect execution to the hook [13]. This can be done with a simple jmp <offset>, but other combinations of instructions can work as well (eg, push <address>; ret). Sometimes, if a function expects to be hooked frequently, it may include several nop instructions at the beginning, allowing space for the overwrite. While inline hooks are simple to understand and implement, detection is fairly easy as well. When hooking a function, you want to gain control at the earliest opportunity, since the idea is to implement your function instead of, or before, the other function. Because of this, inline hooks typically need to be inserted in the first few instructions of the target function, meaning there is a fairly conspicuous control flow change in the first handful of bytes. While changing up the instructions used to divert execution might fool simple tools that scan for ret and the like, any investigator or more robust tool will fairly easily detect this sort of control flow change. Indeed, this is how the Volatility api_hooks plugin detects inline hooks, which will come up in a later chapter.

1.2.2 IAT and EAT

Various executable formats include built-in methods for programs to import and export functions without code being duplicated in memory or invoking the loader each time you want to locate a function. On Windows systems, the PE format makes use of the Import Address Table and Export Address Table (IAT and EAT) [105], which are tables that in the former case store addresses of functions some program wants to import, and in the latter case store addresses of functions that the program (a library, perhaps) wants to export. Similar structures are used on Linux systems, for ELF files, and they are named the Procedure Linkage Table and Global Offset Table (PLT and GOT)[132]. In any case, when a program makes reference to such a function, the lookup is done in the table instead of making use
of the loader to find it. These tables are, however, a prime target for API hooks, because simply overwriting an address in the table hooks the function whose address was stored there. Detecting these hooks is more involved than detecting inline hooks. For instance, Volatility reports functions as hooked if an address in the IAT or EAT points outside of the module that owns that function, unless it matches against a list of known-redirected functions [133].

![Figure 1.1](image)

**Figure 1.1** The function prototype of SetWindowsHookEx.

### 1.2.3 SetWindowsHookEx

*SetWindowsHookEx* [91] is a function exported by the Windows API which installs a given hook for some given event type. The function prototype is given in Figure 1.1. Using this function, programs can add a hook to a chain of hooks to be called, sequentially, once some event specified by the `idHook` parameter happens. The function to be executed when this happens is specified by `lpfn`, and the module in which this function is located is specified by `hmod`. Finally, `dwThreadId` is the id of the thread to be hooked, or zero to hook all threads. Use of this function by malware to install userland hooks has been very popular historically, and still is [49, 127, 54]. This is because other methods of hooking create significant noise on the system, require code injection into other processes, and require polling of keystrokes, which may result in missed keystrokes. The use of code injection and privilege modification by other hooking methods inherently raises suspicions, and at the very least uses more than one system call, while *SetWindowsHookEx* is just one function call, and everything else is handled by the operating system. Since hooks installed this way use a Windows API call, the system must keep track of them somehow. Indeed, Windows keeps track of all information
associated with each hook, and keeps track of each hook associated with each active thread on each desktop [75]. Because of this Volatility can enumerate all message hooks present on a system, as well as their backing modules, which function is called, and which threads they affect.

1.3 Code Injection

Another technique used by malware for an array of malicious behaviors is code injection. Code injection is a broad term for any mechanism by which one process can cause code to be written to, and executed inside of another, victim, process. There are four main techniques malware uses to accomplish this, outlined in the next few sections. All of this information is covered in The Art of Memory Forensics [75], but will be included briefly here for clarity and completeness.

1.3.1 Remote DLL Injection

The first technique discussed is called remote DLL injection. Remote here does not indicate that the injection is being done remotely with respect to the computer being infected, but rather remote with respect to the process being victimized. Remote DLL injection happens when a malicious process writes the path, on disk, of some malicious DLL on disk into the memory of the victim process. Then, the malicious process creates a new thread inside the victim process, and has it call \texttt{LoadLibrary}, passing in the malicious DLL as a parameter. Since the path to this DLL is indeed in the victim processes memory, it is loaded without issue, and now malicious code is executing inside the victim process’s memory. Since this technique uses legitimate Windows APIs to load a DLL that is actually on disk, there are few forensic artifacts left behind by this method, making it difficult to detect. This issue is examined further in Section 2.4.
1.3.2 Remote Code Injection

This technique is more direct than the previous. Instead of writing a DLL’s path into the victim process’s memory and calling `LoadLibrary`, in this case the malicious code allocates memory with read, write, and execute permissions inside of the victim process. Then, malicious code is written directly into that executable page, and a thread is created which begins its execution at the first address containing malicious code. As is the case with the next two methods, this is a noisy endeavor which leaves behind one or more pages of memory which are marked readable, writable, and executable (RWX), a telltale sign that something is amiss.

1.3.3 Reflective DLL Injection

This technique combines elements of the previous two. Like remote code injection, nothing needs to exist on disk to back the memory being written, but instead of some shellcode being written into memory, it is an entire, well-formed DLL. The reason for not having a DLL backing this process on disk is twofold. First, writing to disk leaves behind additional artifacts which can be discovered through other types of analysis. Secondly, not using `LoadLibrary`, which requires a file on disk, means that none of the operating system data structures updated when it is called are touched or created, and so no "official" record of the DLL loading exists. However, as with the previous technique, a memory region with RWX permission must be allocated in the victim process.

1.3.4 Process Hollowing

The last technique differs somewhat significantly from the previous three. In process hollowing, instead of an existing process being victimized to host some malicious elements, a legitimate process is started by the malware, but execution is not started immediately. Instead, the malware first frees all the memory in the newly created victim, then replaces
it with the entirety of the malware, and then the thread executes, beginning now at the malicious code. The effect is that all of the data structures that track process creation hold legitimate process information, but the malware is in fact executing. In order to "replace" the original memory with the malware, however, requires the newly allocated memory to be RWX, so that after allocation the malware body can be written in, and then executed. It shares this similarity with the previous two methods.

1.4 Concrete and Symbolic Execution

As mentioned in section 1.1.2, one can analyze code dynamically in order to study its behavior. One method is to simply execute it and let it run, given the code is in an executable format. Alternatively, working from the static code, be it a disassembly or true source code, one can *emulate* the execution of the code in a virtual environment. Emulation is frequently used in malware analysis and other security-related analysis situations [8, 11, 63, 7, 106, 137]. One tool for emulation is the Unicorn Engine [108], an emulator for several different processors. Benefits of emulating code in this way include the ability to instrument it as much as necessary, assurance that your real system will not be harmed by the code you are executing, and the ability to quickly check different execution paths of the code through different variable assignments. This last point, analyzing the program by tracing execution down different paths by supplying different inputs, is called *concrete execution*. This is particularly powerful because simply executing the code as normal may only drive execution down one particular path in the program, but if you have a way to supply multiple, different inputs, various paths can be explored, giving better information about the true behavior of the code being analyzed. One such way to generate this input is through a *fuzzer* [101, 139], a program that automatically generates different inputs to some given program to try and discover inputs that reach some desired state, usually a crash.

Another method of exploring different code paths involves the use of a technique from
formal program verification methods, called *symbolic execution*. While concrete execution might initialize variables to some given, concrete, state, symbolic execution initializes them to a *symbolic value*. You can think about a symbolic value much like a variable in algebra: it has no value associated with it, but instead has ranges of values that it can or cannot take. In the case of symbolic execution, these ranges are specified in the form of *constraints* which are placed on the path that the program has followed. Take the code in Figure 1.2 as an example. Since there is no way to know what parameters \( f \) is passed, \( x \) and \( y \) would be initialized to symbolic values, say \( Vx \) and \( Vy \). The next step of the execution hits a branch, conditional on the value of \( x \), so two execution paths are explored, one with the constraint \( Vx \leq 10 \) and one with \( Vx > 10 \). The first case returns 1 immediately, ending execution, so it can be seen that in cases when \( \text{param1} \) is less than or equal to 10, the function \( f \) returns 1. The second branch, however, branches again, this time placing \( Vx < Vy \) in the positive branch, and \( Vx > Vy \) in the negative branch. The execution continues on this way until there is nothing left to execute. The end result of this process is that every state of the program is explored, and a set of conditions that must be met to reach those states are generated. When reaching a branch in the program, symbolic execution makes use of a *constraint solver* [51] on each path constraint to make sure that the path can actually be reached, so time is not wasted exploring infeasible execution states.

Concrete and symbolic execution can be applied together, in a process known as *concolic*
Concolic execution initializes input variables to arbitrary, concrete values, but treats all others as symbolic. Additionally, it maintains symbolic state information as well as path constraint information. Because of this, execution proceeds as normal concrete execution down some path. When a branch is reached, then, there is one canonical path that will be taken based on the concrete input variables. Since path constraints are maintained, however, a constraint solver can be applied to find alternative concrete input values that, if provided as input, would result in the other path in the branch being taken. And so, the concolic execution kicks off again with these new values and explores the previously unexplored path, repeating until no new paths are available. The end result of this whole process is that you are left with concrete input values to the target program which can cause execution down every program path.

1.4.1 The Unicorn Engine

As mentioned in the previous section, the Unicorn Engine (or Unicorn) allows for emulation of code on different processor architectures. This work relies on Unicorn to implement HookTracer, and so it will be discussed briefly. Unicorn is based off of another popular emulator, QEMU [10], which is a whole system emulator, a favorite for quickly emulating guest operating systems such as Raspbian [43], and older video game consoles. Unicorn cuts away much of the meat of QEMU, leaving just a light-weight processor emulator. This means that the implementation of everything else is left to the user, including setting up virtual memory, the stack, and any system calls you want to emulate. Because of this, though, Unicorn is very useful for emulating code that was disassembled from a binary or binary data pulled from a memory image. Unicorn also provides helpful bindings in different languages that allow ad hoc execution environments to be set up as part of another program. In the case of HookTracer, Unicorn is integrated into a Volatility plugin in order to analyze hooks detected by Volatility.
1.4.2 angr

This work also made use of angr [116, 121, 115], a binary analysis platform with many different features and applications. In particular, angr was used for its symbolic and concolic execution capabilities in order to get more robust information about the behavior of code. Unicorn could have been used for this as well, but that would have required building an entire symbolic execution engine on top of it, which is in part what angr is already. Due to compatibility issues in Python versions, angr could not be integrated directly into a Volatility plugin, but with future Volatility releases upgrading to Python 3 such integration should be possible.
Chapter Two. HookTracer

2.1 Introduction

The first portion of research conducted for this work produced a framework called HookTracer. HookTracer is an API for Volatility plugins to use which allows them to make use of the Unicorn emulator mentioned in 1.3.1. HookTracer takes care of the subtle work of setting up the stack and providing callbacks for use during emulation, allowing plugins that want to emulate code from memory to be developed quickly and effectively. The research presented in this chapter relied on HookTracer for the development of two plugins, hooktracer and hooktracer_messagehooks, for analyzing API and message hooks respectively, and a third for analyzing injected code, iodine. To avoid confusion, the API developed is HookTracer, capitalized ‘H’ and ‘T’, while the plugin is hooktracer, all lowercase and italicized.

While Volatility’s apihooks and messagehooks plugins are good at discovering the presence of hooks in an image, the sheer number of hooks found on known-clean images of modern operating systems means that determining the presence of a malware infection is both time consuming and tedious, as an investigator must manually examine and reverse engineer the behavior of each hook to decide if it is malicious or not. The same is true for the potentially injected code detected by the malfind plugin. Table 2.1 lists the number of API hooks detected on a clean installation of each of the listed operating systems. This data was collected by installing the 32-bit version of the OS on a virtual machine through VMWare Fusion. The default user was logged in, and then the default web browser was launched, Internet Explorer in some cases and Microsoft Edge in others. The VM was then

suspended, generating \textit{vmem} and \textit{vmss} files. These files were then fed into Volatility, and the \textit{apihooks} plugin was run to enumerate all hooks present. While an investigator might be able to manually check thirty six hooks in a reasonable amount of time, checking 300, or 600 hooks, much less 32,000, is entirely infeasible for even the most experienced reverse engineer. HookTracer allows for the automated analysis of each hook found, with relevant information output for each one, so that investigators can simply run the plugin and get the high-level behavioral information required to make decisions about which hooks are malicious. The following sections will cover the work produced using \textit{hooktracer}, \textit{hooktracer\_messagehooks}, and \textit{iodine}, respectively.

\begin{table}[h]
\centering
\caption{Number of API Hooks by Operating System}
\begin{tabular}{|l|c|}
\hline
Operating System & Number of API Hooks \\
\hline
Windows XP & 36 \\
Windows 7 & 296 \\
Windows 8 & 622 \\
Windows 10 & 32,458 \\
\hline
\end{tabular}
\end{table}

\section{2.2 hooktracer}

\subsection{2.2.1 Input}

The input to \textit{hooktracer} is the output of the Volatility plugin \textit{apihooks} as a JSON file. \textit{apihooks} detects and gathers the API Hooks from the sample as described in Sections 1.2.1 and 1.2.2. Since Volatility is a command line tool, the default output from each plugin is usually directly to the terminal, but there are built in options to have the output as one of several different standard formats and saved to a file. In this case, JSON was decided to be the most useful, as the Python JSON library makes storing output as a data structure and then retrieving the data with the same structure, or a similarly useful one, incredibly easy.
Algorithm 1: Hooktracer

1. $\text{ApiHookSet} \leftarrow$ API hooks from Volatility’s apihooks

2. foreach hook in $\text{ApiHookSet}$ do

3. $\text{BasicBlocks} \leftarrow \text{Emulate(hook)}$

4. $\text{CodeMemRegs} \leftarrow \text{Map(BasicBlocks)}$

5. foreach region in $\text{CodeMemRegs}$ do

6. Display(Region)

end

end

Figure 2.1 An inline/trampoline hook, as displayed by apihooks.

Figure 2.1 shows an example of apihooks’ command line output for a single inline hook.

2.2.2 Emulation

With data about the hook and victim in hand, emulation can proceed. For this emulation to yield any valuable information, callbacks need to be registered for a variety of occurrences. At the time this research was conducted, HookTracer had callbacks for the following events:

- New instruction reached
• New basic block reached
• Memory reads and writes
• Memory accesses (read, write, or execute) to invalid or unmapped memory regions

After the callbacks are registered, a virtual address space is initialized, the first part of which is the stack. Since the stack needs to be maintained inside of the same virtual address space that the rest of the emulation is happening in, its addresses need to be picked carefully so that nothing the code being emulated does overwrites stack data. To this end, HookTracer implements its stack in a memory region reserved for the kernel on Windows systems. This region is chosen because userland code can never access kernel ranges, so the stack will be safe from any operations executed. An additional consideration related to the stack is that finding a way to determine when the hook is finished executing. Since the idea is to execute the whole hook, execution cannot be stopped at a predetermined depth, but similarly executing code unrelated to the hook yields incorrect results, or unbounded execution. To address this, the read and write callbacks keep special track of the base address of the stack. Since the stack is initialized with no data, a flag is set False, marking the stack empty. The memory write callback includes special handling so that when the stack base is written to the flag is set to True, meaning that the code being emulated resulted in something being added to the stack, for instance because of a function call. When the last piece of data is popped off of the stack, the flag is set back to False, and so on. The memory read callback includes special handling so that when the stack base is read while the flag is set to False, the emulation terminates. This is because when the hook executes its final `ret` to return control back to the calling function, the stack will be completely empty because all subroutines will have finished executing, and the `ret` will point to the base, because in a real execution environment the base we set up would contain the address of the original function that called the hook.

Now that the emulation environment is set up, emulation can begin. This starts at the
initial address of the hook, provided by the input file. Since the virtual address space is initialized empty, no matter what the address is, it will trigger a callback set to handle an access to invalid memory. This callback makes sure that the address is valid in the address space of the process being analyzed, and reads in a block of memory starting at that address memory image, if it exists. If it does, the data is copied into the emulated address space and emulation continues. This callback is also triggered when control is passed to an unmapped region in the emulated address space. This way, the emulated address space is filled in on demand as emulation proceeds. This is important since the code being emulated is only kept track of by its starting address, it is impossible to know ahead of time exactly what memory will be accessed. When the page requested is not resident in the memory image, the plugin can attempt to help the emulation along with behavior primarily focused on allowing the hook to execute for as long as possible. In the case of writes, an empty page is mapped into the emulated address space at the address requested, and the write continues. For a call to a non-resident page, the callback maps in a page at that address filled with the opcodes for \texttt{mov eax, 0; ret}, so the hook believes that whatever function it called returned 0 and behaves accordingly, instead of simply halting emulation.

2.2.3 Gathering Basic Blocks

The callback registered to trigger on new basic blocks is what allows for most of the high-level analysis to be conducted. Each time the callback is triggered for a particular hook, the starting address of the block is recorded. After emulation terminates, these blocks are checked, using Volatility, against the memory image to find what memory region contains them. The resulting list of memory regions, in the order they were reached, gives high-level data about the execution, allowing for the possibility of a variety of analyses.
2.2.4 Analysis

The most basic analysis allowed for by hooktracer is reasoning about memory regions and permissions. Figure 2.2 shows a snippet of the output from hooktracer when run against a memory image infected with the Coreflood [67] malware. The first information displayed comes directly from apihooks, that a process, IEXPLORER.EXE with PID 2044, has a hook installed on GetMessageA in the DLL user32.dll. All of the information presented after this, a list of triples containing a set of permissions, the memory region, and the number of basic blocks executed in that region consecutively, was generated by hooktracer. The memory regions listed were pulled from a kernel data structure called a Virtual Address Descriptor (VAD), which means it cannot have been overwritten by a userland process attempting to hide [6]. In this example we see that control is first redirected to a non-file-backed memory region starting at the address 0x7ff80000, which is marked readable, writable, and executable. This should immediately raise alarms for investigators, as legitimate code would normally be backed by a file on disk instead of executing directly from memory. Additionally, a memory region being readable, writable, and executable (RWX) is bad practice, and as such is usually the result of malware changing permissions so that it can inject code into memory directly and then execute it. Cysinfo [1] details how DLLs loaded legitimately will have the permissions PAGE_EXECUTE_WRITECOPY, while hollowed processes will usually have something else, most commonly PAGE_EXECUTE_READWRITE. After code executes in these non-file-backed regions, control is passed to the legitimate handlers for the request, then to the malicious code, back to legitimate software, and finally back to the malicious code again.

A second piece of malware, TDSS [94], was examined with hooktracer, and a snippet of that output is presented in Figure 2.3. In this case, the malware starts executing in a malicious region located at 0x270000, then jumps to yet another malicious region located at 0x260000. From this an investigator can identify two separate regions hosting suspicious
As mentioned in Section 1.2, antivirus software makes extensive use of API hooks to monitor system behavior. Unfortunately, this compounds the problem of false positives. Taking the clean Windows 7 install used for the data in table 2.1 and installing a free edition of AVG Anti-Virus [110] increased the number of hooks from 296 to 1,625. Clearly this is problematic for investigators, even more so because the hooks installed by antivirus software frequently use the same tricks employed by malware. For example, based on the previous two malware examples one might assume that any execution from a non-file-backed region with read, write, and execute permissions is malicious. While this is often the case, consider the results of running hooktracer on the clean Windows 7 image with AVG Free Edition, shown in Figure 2.4. Just like the malware seen previously, AVG executes the first part of its hook from a non-file-backed RWX memory region before jumping into several DLLs owned by AVG. Based on the output of hooktracer, then, an investigator can perhaps make
the decision to ignore hooks such as this one for now, since they are likely the result of well known anti-virus software, and instead focus efforts elsewhere. Again, this is accomplished with no reverse engineering on the part of the investigator.

Figure 2.4 AVG API hook detected by hooktracer.

While hooktracer does significantly lessen the load on investigators, turning thousands of reverse engineering efforts into the same amount of processed output still leaves a lot on their plate. To address this, hooktracer supports the use of three types of filters. The first type, called an All Containing filter, filters out any output where all of the memory regions visited match a given file or directory. For instance, filtering out all the hooks where every basic block was executed inside the System32 directory was found to reduce the number of hooks outputted on the clean Windows 10 image used in Table 2.1 from 32,458 to 178. This can be done because on modern Windows systems, the DLLs in the System32 directory are protected from modification. Furthermore, analyzing these remaining 178 hooks revealed an additional pattern: many of the remaining hooks execute exclusively in directories involving the Visual C++ runtime system and Microsoft Office’s One Drive, as shown in Figure 2.5 and Figure 2.6. By adding All Containing filters for these directories as well, the number of outputted hooks dropped further from 178 to zero.

A second filter type, the Any Containing filter, is a kind of complement to the first. An Any Containing filter filters out all outputs where the path of any memory region matches a file or directory given. This is useful in cases like the AVG example above. If an investigator decides she does not want to see any hooks related to antivirus, she can supply an Any
Figure 2.5 An unfiltered API hook related to use of the `vcruntime140_app.dll` in the WindowsApps folder.

Figure 2.6 An unfiltered API hook due to use of Microsoft’s OneDrive components.

Containing filter to make it so. As shown in Figure 2.4, all of the hooks inserted by AVG are in the `\ProgramFiles\AVG\Antivirus` directory. Testing this exact filter, along with the previously described All Containing filter, on the Windows 7 VM with AVG Free Edition installed reduced the number of displayed hooks from 1,625 to 175, lower than the 296 from before AVG was installed.

The last filter type, Grouping, causes hooktracer to only output hooks matching a specific pattern. The power of this capability can be demonstrated through tests on the infamous Zeus malware [59, 134]. The malware was installed on the previously clean Windows 7 system, without AVG installed. Because Zeus attempts to hook 41 functions in every process it can access, this brought the total number of API hooks up to 480 from 296. Without any filters on, many of the hooks follow a similar pattern, shown in Figure 2.7. This is the output of hooktracer for one hook, with the permissions section removed for readability. The control moves from one suspicious region to another, then into a legitimate region owned by System32, and then finally back to the first region. All of the hooks installed by Zeus follow this pattern, but with different numbers of DLLs in System32 being used, and different addresses on the anonymous memory regions, due to different memory layouts for different processes. A grouping of hooks that look like this can be generated by hooktracer’s grouping
functionality. This filter is generated by specifying a function name and PID of the hook to serve as a template as parameters to \textit{hooktracer}. \textit{hooktracer} then records, in order, the first three memory regions executed by the hook, making note of the size (not the addresses) of the anonymous regions and the full path of the file-backed regions. This record is saved, and if the investigator runs \textit{hooktracer} again, supplying the record, only hooked functions that match the record are displayed. Figure 2.8 shows the (truncated) result of this process. By specifying one function to serve as an example, all of the hooks, and only the hooks, generated by Zeus are displayed. An additional benefit to this type of grouping is that the record is saved to disk, meaning it can be used in future investigations to quickly check for Zeus infections. An investigator can then generate a hook signature for each type of malware she comes across and incorporate them into her automated workflow or share them with others, allowing her knowledge and experience to propagate efficiently.

\begin{verbatim}
2384 taskhost.exe ntdll.dll!NtCreateUserProcess at 0x79f5778
<Non-File Backed Region: 0x850000 0x87bfff>
<Non-File Backed Region: 0x9a0000 0x9a0fff> (2)
\Device\HarddiskVolume1\Windows\System32\ntdll.dll (2)
<Non-File Backed Region: 0x850000 0x87bfff> (5)
\end{verbatim}

\textbf{Figure 2.7} \textit{hooktracer} output for a Zeus API hook.

\section*{2.2.5 Limitations}

While \textit{hooktracer} does alleviate much of the reverse engineering efforts previously required of investigators, it does not, by itself, give any indication about the legitimacy of a certain hook. Furthermore, while the filters are a useful tool, filtering is a fairly brittle process. For instance, if malware makes an effort to specifically infect antivirus software it may get caught in filters because it looks legitimate. Lastly, only seeing the memory regions visited and not the specific behavior of the hook can make it hard to make determinations about hooks that make it past any filters. Techniques developed for use in plugins developed after \textit{hooktracer}, presented in the following sections, address this last issue by allowing for the
capture of specific API or system calls found during emulation.

### 2.3 hooktracer_messagehooks

The second plugin, *hooktracer_messagehooks*, acts on the output of the *messagehooks* plugin, and thus the behavior of message hooks on a system instead of API hooks. This required the development of new functionality for HookTracer in the form of callbacks for different behaviors, as well as additional input handling and post-processing. Specifically, *hooktracer_messagehooks* is designed to target keylogging malware, the main abuser of *SetWindowsHookEx*, but functionality could be expanded to handle message hooks installed for other malicious reasons. In particular, the number of instructions which need to be executed to emulate message hook behavior greatly exceeds what was required for API hooks, and the use of system calls by message hooks is again much greater than that of API hooks.
2.3.1 Input

Like hooktracer, hooktracer\_messagehooks relies on the output of another Volatility plugin to function, messagehooks. The usage and basic principles behind messagehooks is described in Section 1.2.3, and Figure 2.9 shows a snippet of this output for a global hook detected on a system, while Figure 2.10 shows the same hook from the local context of a specific thread. Figure 2.9 shows that the hook is global, indicated by the line where the thread is listed as "<any>." This means that the hook is installed on any thread active on the desktop logon session indicated in the "ihmod" line. In this case, 0 means that the hook is active in the Default desktop. The hook listens for WH\_CALLWNDPROC messages, indicated by the "filters" line, and the function to be executed is located at offset 0x1160 in C:\Windows\system32\wls0wndh.dll, specified next to "Procedure" and "Module," respectively. Figure 2.10 shows further that this hook is active in thread 3180, in the process spoolsv with PID 1360. One block such as this is reported by messagehooks for each thread running on the desktop, since the hook requested to be active in all threads. Again, as with hooktracer, the output from messagehooks needs to be in the form of a JSON file.

```
Offset(V) : 0xfee009d8
Session : 0
Desktop : WinSta0\Default
Thread : <any>
Filter : WH\_CALLWNDPROC
Flags : HF\_GLOBAL
Procedure : 0x1160
ihmod : 0
Module : C:\Windows\system32\wls0wndh.dll
```

Figure 2.9 A global message hook as displayed by messagehooks.

2.3.2 HookTracer Augmentations

hooktracer\_messagehooks, like hooktracer, makes use of the HookTracer API for emulation. As such, much of the basic set up is taken care of as described in Section 2.2.2. For message hook emulation to be effective, though, more functionality needed to be added to HookTracer.
Since message hooks rely more heavily on the operating system to provide services and access to resources than API hooks, in general, simply executing the code without the operating system to support it does not get very far. Therefore, HookTracer was augmented with implementations of Windows APIs that will work in the virtual environment. The main behaviors targeted for implementation were:

- Lock access - Since no other processes or threads are active within the emulated environment, the state of a lock is "stuck" at its value at the time of memory capture

- Memory region allocations - Handling of memory region allocation requests, such as through the `VirtualAlloc` API, requires in-kernel code that is not supported by the emulator

- Debugging APIs - These APIs allow reading and writing memory of other processes, none of which are present within the emulated environment

In order to make sure calls to these API functions are properly handled, before emulation is attempted `hooktracer_messagehooks` scans the hook process for all loaded modules and records the address of all functions exported by these modules. When one of these functions matches one implemented in HookTracer, any calls to these functions are instead directed to the HookTracer version.

Aside from differences in operating system usage, API hooks and message hooks generally behave differently. API hooks typically record or manipulate parameters being passed to whatever function they hook, while message hooks typically have more complex behavior,
such as filesystem, registry, or network activity. Activities in these domains can be used for persistence, propagation of infection, stealing data, or spying, so it is important HookTracer is able to monitor them. Even though a goal of this work is detection of malware that does not interact with the filesystem, it would be negligent to ignore such artifacts when they exist.

In order to monitor filesystem activity, HookTracer was expanded to maintain an internal table of handles, unique identifiers that HookTracer creates and passes back to calls such as OpenFile. On reads and writes to a specific handle, HookTracer makes use of Volatility [133] to attempt to read a cached version of the file in memory. In the case that the file is not in memory, a buffer of null bytes is returned instead. Attempted writes to a file are instead directed to a file that shadows the original, with new data overlayed on the original data. Lastly, a close operation causes HookTracer to drop the handle from its internal table. Additionally, calls to functions that return file metadata, such as ZwQueryInformationFile [93], are commonly used and so are partially implemented by HookTracer such that most common uses of the function are supported. Similarly, GetFileSize is implemented by returning the size of the file if it is found cached in memory, or else 256KB is returned.

Similar to the file system, accesses to the registry also need to be handled, and as before, HookTracer maintains a virtual registry tree containing keys and values pulled from memory when possible [133] or generated when needed. This again allows the malware to be given as much real data as possible to act on, which in turn gives hooktracer_messagehooks correct information about the malware’s malicious behavior.

A common feature of malware, especially memory-only malware, is reliance on networking. In some cases, malware might not execute at all if it cannot connect to some website or server. To this end, HookTracer provides as much fake network connectivity as possible. This is a known problem in the security community, especially for attempts at dynamic analysis. With malware being, well, malicious, it becomes very risky to actually give an infected system access to a network and so malware analysis sandboxes are isolated. To
get around this, projects such as FakeNet [53] and its successor, FakeNet-NG [41], provide fake DNS and HTTP services to whatever sandbox an investigator connects them to. This way, malware executing in the sandbox believes it actually can connect to the Internet and executes normally. Unfortunately, HookTracer cannot use these tools directly, but instead makes use of similar techniques to fake network activity and connectivity. For instance, DNS resolution attempts cause HookTracer to return Google’s public IP address. Calls to functions that perform HTTP requests cause HookTracer to generate a reply that matches the requested protocol and file type. Calls that expect raw data to be returned cause HookTracer to generate the English alphabet, repeating to match the expected message length. Lastly, calls that expect no data but attempt to interact with network devices, such as `bind`, `accept`, and `socket`, simply cause a return of a value that indicates success.

Returning fake data is a method which has come up a few times now, and which is used for several other functions where trying to implement the function does not, or may not always, work. To successfully fake returned data, the register holding the return value must be set (EAX or RAX for 32 or 64-bit systems, respectively), and filling in data structures the calling code expects to be filled:

- **GetComputerName[83]:** Malware commonly wants to gather information about the system it has infected, including the name, for purposes of associating stolen data with a particular computer. Additionally, this is sometimes used to check if the malware is being executed in a VM, by parsing the name for "VM" or "Virtual Machine," as is sometimes the default. To handle this function, HookTracer always fills the given buffer with "Windows7Desktop."

- **GetModuleFileNameW[85]:** Malware uses this function to find the full path to where it is running from disk. In this case, while HookTracer fakes the function, the data returned is correct. HookTracer uses the Volatility API to get the path of the module requested from the memory image and returns that in the given buffer.
• **System Time Functions**: Functions in this family include `GetSystemTime` [86], which returns the current time, and `SystemTimeToFile` [92], which converts the time supplied by `GetSystemTime` to a human-readable standard. Malware can use system time functions for a variety of reasons, from simply logging time of infection to attempting to determine if the environment is virtualized, which may be the case for some strange timestamps. To provide sane values, HookTracer gets the system time from when the memory dump was taken using Volatility’s API and returns it.

• **Functions used for Timestomping**: Timestomping [31] is a name given to an anti-forensics technique where malware attempts to blend in to the rest of the system by altering the timestamps of its own files on disk. First, `GetFileTime` [84] is called, targeting a common system file. Then, with the normal-looking timestamp in hand, the malware calls `SetFileTime`[90] to copy this information onto the malicious file’s timestamps. To handle this behavior, HookTracer returns a sane timestamp, as in the case of `GetSystemTime`, and returns a "success" for the overwrite. It is worth noting that if such behavior is emulated, the malware may point the investigator directly to the very files it was trying to hide.

• **Filesystem Enumeration**: Functions that allow for traversal of the filesystem, such as `FindFirstFile`[81] and `FindNextFile`[82], are often used by malware for finding files. For instance, an effective piece of ransomware is one that does not leave any important files alone, and as such wants to traverse the whole `Pictures` or `Documents` directory. To accommodate such malware, HookTracer fakes a realistic directory structure.

• **Process Enumeration**: Malware may want to scan the list of currently running processes to find victims for code injection, checking for security software, or checking for VMware tools running on the system to detect virtualization. Functions that allow for process enumeration are `CreateToolhelp32Snapshot` [80], `Process32First` [88], and
Process32Next [89]. To deal with these calls, HookTracer forges the data structures required and uses Volatility’s list of active processes to return real information.

The last major revisions that needed to be made for HookTracer to accommodate message hooks involved functions. For emulating API hooks it was enough to gather low-granularity information about the memory regions where basic blocks executed, and to generally help functions along when possible. For message hooks, however, specific function parameters and data structures matter more for correct malicious behavior, and the specific functions called inside of a DLL, in addition to the order they are called, matter more as well. To handle the issue of specific parameters, HookTracer fakes function parameters to certain functions before emulating them. As an example which will be detailed in the next section, HookTracer sets up parameters and data structures that Windows would normally generate after a keystroke to ensure keyloggers can act properly on the data. In order to get the requisite detail on which functions are called and how they are used, HookTracer records the parameters passed and the function names, instead of just recording the memory region. In order to accomplish this, HookTracer is passed a list of functions to monitor where each call to any of those functions triggers a callback which can then get the specific function name, as well as the parameters passed to it, from the call.

2.3.3 Analysis

![Figure 2.11 Function prototype for a message hook handler.](image)

With the API modifications documented, discussion of hooktracer_messagehooks itself can begin. The plugin first needs to manually create the data structures that would be created by Windows during a keystroke. Figure 2.11 shows the parameters sent to the message hook handler (the ones that need to be faked) on each event. First, nCode holds either
HC_ACTION or HC_NOREMOVE. Both indicate that the rest of the parameters hold keystroke information, with HC_ACTION indicating that the keystroke is current. Next, wParam holds information associated with the event. Typically, keyloggers will look for WM_KEYDOWN events, as shown in a snippet of the Gozi [119] malware leaked on github [14] in Figure 2.12. Lastly, lParam holds a flag which describes the (device specific) key value and other attributes [87]. When populating these fields on creation of the data structure, the plugin fills in the values as if the user typed ‘A.’ Then, using HookTracer, these data structures are mapped into the emulated address space. Now, during emulation, any hooks triggered on a keystroke will actually have data to read, and thus malware can proceed with execution of its payload.

```
if (!g_bLoggerEnabled || (nCode != HC_ACTION) || !HookStruct ){
  break;
}

if ( (UINT)wParam != WM_KEYDOWN ){ //message
  break;
}
```

Figure 2.12 Leaked Gozi source code.

In order to determine if a hook is malicious or not, hooktracer_messagehooks looks for any of several behaviors. At a high level, what most strongly differentiates legitimate and malicious hooks are how long it takes to process the keystroke data, and where the data is kept. Legitimate processes are usually fairly quick and do not move the data out of memory. With that in mind, the following are flagged as strong indicators of malicious behavior:

- Keystroke data is logged to a file, the network, or the registry
- The clipboard is accessed
- Screenshots are taken
- Debugging APIs are used
- CallNextHookEx is called immediately
The first indicator is flagged because secure, legitimate software will very rarely store keystrokes at all, but keyloggers exist to get that information off of the infected system somehow. Clipboard access is flagged because spyware frequently grabs the contents of the clipboard on keystrokes to try and get passwords which may have been copied. Similarly, spyware might take screenshots during keystrokes with hooks inserted into the web browser to get information such as usernames or bank account values. The previously mentioned Zeus malware does this. Debugging APIs are used by malware for code injection, or for inspecting the hooked process’ memory. Lastly, an immediate call of `CallNextHookEx` is flagged. This function is called at the end of a hook to tell the system that the current hook is finished, and to call the next one in line. However, since `SetWindowsHookEx` injects an entire DLL into the hooked process, malware can use it just to inject a malicious DLL into many processes by registering a hook that just calls the next hook. Laqma [73] and Carberp [66, 98] used this technique.

Similar to the grouping functionality in `hooktracer`, `hooktracer_messagehooks` supports trace records. In addition to marking hooks as malicious or not, `hooktracer_messagehooks` can list, in order, the specific activity of a hook via the APIs called, and optionally the parameters for the functions passed to HookTracer in the aforementioned array. This list, called a trace record, helps an analyst see, in detail, the high level behavior of the hook. The analyst can pass one or more trace records corresponding to known malicious hooks to `hooktracer_messagehooks` to automatically identify these hooks in different processes in the same system, or on entirely new systems in the future. The goal, as with groupings, is to allow knowledge and experience to easily propagate.

In order to demonstrate the capabilities of HookTracer and `hooktracer_messagehooks`, it was tested against several real malware samples. First on the docket was Turla [32, 70, 22]. Turla is the name of an advanced persistent threat (APT) group thought to be an elite hacking group inside of one of Russia’s intelligence agencies, but also refers to their platform for digital espionage. Among many other capabilities, Turla provides the ability to
log keystrokes and other system data. In particular, Turla makes use of `SetWindowsHookEx` to collect this information [76, 77].

Turla was executed in a new VMWare Fusion VM running 64-bit Windows 7 Professional, updated for all of the latest security and operating system versions. The sample executed has MD5 hash:

```
59b57bdabee2ce1fb566de51dd92ec94
```

This is a DLL deployed by the Turla platform. `rundll32.exe` was used to execute the DLL as a traditional executable, a technique documented in recipe 13-2 of the Malware Analyst’s Cookbook [74]. After execution, to provide normal system usage Turla would monitor, Notepad was launched and a sentence typed. Next Internet Explorer was launched, navigating to Google. Lastly, after waiting for 30 seconds, the VM was suspended, causing a memory dump to be emitted, as described in Chapter 1.

The `hooktracer_messagehooks` plugin defaults to simply emitting only message hooks it determines are malicious. To get a better idea of the behavior of these hooks, an investigator can supply the command line option `-list-apis`, causing the plugin to generate an ordered list of exported functions called by the hook, as well as their parameters. This list can be instead filtered down with the option `-list-apis-condensed`, causing the list to only show functions that match a built-in filter. Figure 2.13 shows the result of running the plugin with this last option against the hook installed by Turla.

The output is fairly lengthy since Turla generates significant activity with its hook. First (lines 1-5), Turla collects a handle to the current window and the system time and makes them into a single string. Next (lines 6-10), Turla gathers and formats the process ID followed by the name of the file backing that process. Lines 11 and 12 show Turla getting and formatting the Window Text, which is faked by HookTracer to be "MyWindowName." The next line shows Turla moving all of the previous values together with `memmove`. Lines 14-18 show Turla turning the device specific keyboard code into its corresponding letter, in
Figure 2.13 *hooktracer_messagehooks* against Turla.
this case the ‘A’ faked by HookTracer, and then formatting it into a string. Lines 19-26 are a bit more complicated, but the overall behavior is that Turla is building a path to the file where it will store the gathered information, and then opening a handle to that file. In this case the handle is faked by HookTracer (41424344). Lines 28-32 collect and format information about the name of the computer and the logged on user, and then concatenate it with all of the information previously gathered, finally, in line 33, writing it to the file. Lastly the handle is closed and the next hook is called. It is important to stress that this information required absolutely no reverse engineering on the part of the investigator, and very minimal knowledge about Windows APIs (most of it is in the names of the strings). And yet, the investigator can verify very quickly for herself that hooktracer_messagehooks was indeed correct in its categorization of this hook, and it is made explicitly clear to her what the malware is doing. In total, analysis of each hook on the system took less than a minute on a Debian VM with 2 CPU cores and 2GB of RAM. Note that in this output is a path to a file where the logged information is stored. Information such as this, references to the host system by functions interacting with the filesystem, registry, or network, can be automatically saved by hooktracer_messagehooks for future reference or investigation.

Figure 2.14 Trace record for Turla.

Figure 2.14 shows the trace record corresponding to the hook in 2.13. It includes essentially all of the same functions with the exception of string and buffer manipulation functions. These functions, such as sprintf, can be easily swapped out as the malware is updated, and so would make the trace record more brittle. Instead, the trace record only includes Windows APIs which change very infrequently, and would be difficult to replace with other calls and get the same behavior.
The next malware tested against was Loki Bot [79, 125], spyware that focuses on stealing credentials and logging keystrokes. Unlike Turla, Loki Bot is not a nation’s espionage tool, but is instead for sale on underground markets. The version tested has MD5 hash:

```
eccad903b4c27d149e159338f58481a9
```

and is a 32-bit executable. In order to correctly execute it, the steps given in [103] are used. Like with Turla, Notepad was opened and used to create a short document, and then Internet Explorer was opened, navigating to Google. After 30 seconds the VM was suspended. Figure 2.15 shows the output of `hooktracer_messagehooks` with the `-list-apis-condensed` option.

Much of the behavior here is the same as with Turla. The first 16 lines are used to retrieve the keystroke value, line 18 retrieves the (faked) window name which it writes to a file. Line 30 shows the hook reading the contents of the clipboard (faked to be "BBBBCCCCDDDD"), re-opening the file, and writing the contents of the clipboard prepended with ‘CB.’ Lastly, the faked keystroke, ‘A,’ is written to the file before the hook passes along execution to the next hook.

Lastly, the plugin was run against Gozi, also known as Ursnif or ISFB, a banking trojan which is still widely used after over a decade, albeit with many modifications [112, 120, 114]. The sample executed was a 32-bit executable with MD5 hash:

```
F e6d118192fc848797c15dc0600834783
```

Like the last two samples, Notepad and Internet Explorer were run and interacted with, and then 30 seconds later the VM was suspended. Figure 2.16 shows the output of the plugin with the condensed API listing option. First, Gozi consumes the infected thread’s input data, collects information about the module it has infected, collects system time and current window, and collects information about the keystroke. In addition to performing operations on the keystroke, Gozi also makes use of debug APIs by interacting with the thread input queues, hitting another of the malicious criteria.
1. LoadLibraryW: user32 -> 77200000
2. GetForegroundWindow
3. LoadLibraryW: user32 -> 77200000
4. GetWindowThreadProcessId
5. LoadLibraryW: user32 -> 77200000
6. GetKeyboardState
7. LoadLibraryW: user32 -> 77200000
8. GetKeyboardLayout
9. LoadLibraryW: user32 -> 77200000
10. ToUnicodeEx
11. LoadLibraryW: user32 -> 77200000
12. GetAsyncKeyState
13. LoadLibraryW: user32 -> 77200000
14. GetKeyState
15. LoadLibraryW: user32 -> 77200000
16. GetForegroundWindow
17. LoadLibraryW: user32 -> 77200000
18. GetWindowTextW
19. LoadLibraryW: user32 -> 77200000
20. wsprintfW
21. wsprintfW
22. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C8383\34A837.kdb
23. SetFilePointer
24. GetProcessIdOfThread
25. WriteFile: A1424344 -> 48 -> Window: MyWindowName
26. CloseHandle
27. LoadLibraryW: user32 -> 77200000
28. OpenClipboard
29. LoadLibraryW: user32 -> 77200000
30. GetClipboardData
31. GlobalLock
32. LoadLibraryW: user32 -> 77200000
33. wsprintfW
34. wsprintfW
35. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C8383\34A837.kdb
36. SetFilePointer
37. GetProcessIdOfThread
38. WriteFile: A1424344 -> 28 -> CB: BBBBBCCCCDDDD
39. CloseHandle
40. GlobalUnlock
41. LoadLibraryW: user32 -> 77200000
42. CloseClipboard
43. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C8383\34A837.kdb
44. SetFilePointer
45. GetProcessIdOfThread
46. WriteFile: A1424344 -> 2 -> A
47. CloseHandle
48. LoadLibraryW: user32 -> 77200000
49. CallNextHookEx

Figure 2.15 *hooktracer*_ **messagehooks** against Loki Bot.
2.3.4 Limitations

By attempting automated detection of malicious hooks, hooktracer_messagehooks becomes somewhat brittle. For example, the yty malware used by the Donot Team [124] APT is a
keylogger that uses `SetWindowsHookEx` but does not get flagged as malicious by the plugin. Figure 2.17 shows the result of running the plugin on a `yty` hook. The malware does not try to log the keystroke to a file, only allocating and deallocating some memory and performing conversions from keystroke to character. As it turns out, the keystrokes are stored in memory until a certain number are accumulated, at which point they are written to a file, but since HookTracer only fakes a single keystroke, this behavior is missed. While this technique of holding onto keystrokes before writing them to disk is fairly rare, as it can cause missed keystrokes, it does demonstrate that some malicious behavior can be overlooked by HookTracer. For example, a hook handler that only behaves maliciously if a certain sequence of keystrokes are detected, a username for instance, would be very hard to trigger. In these cases, however, the plugin still tells an investigator that this executable handled the keystroke, which may lead them to investigate further since it is not readily apparent what is happening. This issue is hard to overcome through the methods discussed thus far, but a possible way forward is discussed in Section 4.1.

Figure 2.17 `hooktracer_messagehooks` against Donot Team’s Keylogger.
2.4 Iodine

In Section 2.2.4 it was mentioned that a hook passing control flow to a section of memory not backed by a file on disk, an injected region, may indicate malicious activity. Indeed, code injection is frequently employed by malware for a variety of reasons, including staging code for hooked API functions. Since this technique might be used even when no code hooks are used, it is useful to detect injected code separately from code hooks. The Volatility plugin \textit{malfind} does just this, detecting code injected through remote code injection (1.3.2), reflective DLL injection (1.3.3), or process hollowing (1.3.4). Similar to \textit{apihooks}, however, \textit{malfind} suffers from a relatively high false positive rate, and produces a significant amount of output which does not tell the investigator much of use about any injected code. Figure 2.18 shows the truncated output of \textit{malfind} when run against a memory sample infected with Stuxnet [40]. This view allows the investigator to see some useful strings, identify possible shellcode, and locate the process and address of interest. However, \textit{malfind} gives many such outputs, and there is clearly a significant amount of follow-up work to be done on each. Due to this problem’s similarity to the problems that lead to the development of HookTracer, modifications were made again to the API in order to support development of a plugin, \textit{iodine}. This plugin consumes the output of \textit{malfind} and attempts to discover and emulate injected shellcode. The following sections detail the changes made to the HookTracer API to accommodate this plugin, introduce the test data, and explore the results of applying these methods to injected code.

2.4.1 Once More, With Feeling

Several changes and modifications needed to be introduced to HookTracer in order to accommodate the new challenges. Firstly, while previous iterations included functionality for noting exported function addresses to resolve them to symbol names, this update took this process a step further. Not all exported functions are discoverable by walking the module
Figure 2.18 Process infected by Stuxnet, as uncovered through *malfind*. 
list exposed by Volatility, as some DLLs only provide this information in a related Program Database (PDB) file. For completeness, when this portion of the work was beginning, HookTracer was updated to check for the existence of a file containing data processed from a PDB file corresponding to each DLL in the module list and adds the names and addresses of functions in that file to its exported functions list. To produce such a file, the project pdbparse [36] can be leveraged to automatically download the PDB associated with a specific DLL from Microsoft’s Symbol Server. Then, the script pdbparse_print_gvars can be used to parse the PDB into a format consumable by Hook Tracer. This technique will be used again later, in Section 3.2.2.

As with hooktracer_messagehooks, the HookTracer API needed to be augmented to address problems specific to consuming malfind output and exploring shellcode. Unlike the output of apihooks, some regions detected by malfind are not suitable for emulation. Often a region will be detected which is mostly full of zero-bytes. Such a region was likely reserved by a process for legitimate reasons, and attempting to emulate it can lead to unexpected behavior or misleading results. Similarly, as discussed in Section 1.3, entire PE files may be mapped into a victim process’s memory. While this is indeed maliciously injected code, emulating the PE the way we emulate code stubs will not yield useful or correct results. To work around this, HookTracer’s callback for accesses to unmapped memory was changed to detect executable headers at the start of the page, or to detect pages that start with a large number of zero-bytes. If either of these things are true, the page is not mapped into memory, halting emulation, and an error is emitted explaining the situation. This process cuts down on the run time of the plugin and produces cleaner results.

The next hurdle was detecting actual shellcode. Another type of false positive frequently encountered through malfind is legitimately mapped RWX memory inside of a process which contains benign code, or perhaps junk data which disassembles into nonsense code. Malicious code, however, will not only be valid assembly, but will often attempt to quickly pivot to some other region in memory. To differentiate this from benign or junk code, HookTracer’s
basic block discovery callback was changed to parse out instructions which may redirect control flow, such as jumps or calls. The targets of these calls are then either pulled from the instruction, in the case of branches to some immediate, or calculated based on the values in referenced registers or memory. After all the branch targets in a given basic block are discovered, a check is done to see if both the target of the branch and the branch instruction lie inside the VAD where emulation started. In cases where the instruction is inside the starting VAD but the target is outside, the callback flags the branch to be reported to the investigator. Additionally, for clarity, the branch target is compared to list of exported functions HookTracer maintains for the emulated code. If the branch targets an address near the starting address of an exported function, the branch is reported as targeting an offset from that function. This will oftentimes highlight code hooks immediately, because when the malicious activity of a hooked function is complete, it typically returns to the original implementation to finish processing the API call correctly. This is the other side of a phenomenon noted in Section 2.2.4, and why hooktracer was designed to report jumps to non-file-backed regions.

Lastly, more functions needed to be "implemented" in HookTracer, as described in Section 2.3.2. These were all functions discovered through iteratively running the plugin, checking which functions caused strange output, and writing implementations to try and alleviate the issues.

2.4.2 The Usual Suspects

In order to test iodine, a set of memory images from computers infected with a variety of malware was used. This set was provided as part of the Volatility Malware and Memory Forensics Training course, and was chosen because the infections were known and studied by expert investigators, and a write up on each of the infections, in the form of a Lab Guide, existed and allowed for a ground truth to be established in terms of correct detections, false negatives, and false positives. Of the memory images in this set, nine were infected with
malware which actually performed code injection, and so these were specifically used to test *iodine*. The included malware were:

1. Stuxnet, Windows XP: A computer worm which infected Windows systems in an attempt to eventually compromise industrial control systems [40].

2. Rovnix, Windows 7 x64: A bootkit which adds the compromised computer to a large botnet [58].

3. Infected PDF, Windows XP: A malicious PDF file, not associated with a known malware family, which exploits an (at the time) zero day in Adobe Reader. It has SHA-1 hash: d56583b014ccbefc42c9f507a0865d4f5c825f5b.

4. Zeus, Windows XP: A banking trojan discussed in Section 2.2.4 [59, 134].

5. Mimikatz, Windows 8 x64, Windows Server 2008 x64, Windows 7 x64: Mimikatz is an open source tool for credential stealing [35]. While this tool is frequently employed by penetration testers, it is also used in actual malware. Four of the memory images were infected with mimikatz, one Windows 8 system, a Windows Server 2008 system, and two Windows 7 systems. The last three were part of the same scenario meant to model an APT moving across a corporate network.

6. TDSS, Windows XP: A banking trojan and bootkit discussed in Section 2.2.4 [94].

Once all of the memory images were collected, *malfind* was run against each sample, with output specified as a JSON file. Then, *iodine* was run, targeting the corresponding sample and JSON, with the output being directly written to a text file.

2.4.3 Analysis

As described in Section 2.4.1, analysis was an iterative process, where results from one run of *iodine* were used to update the API in useful ways. Here, we discuss the analysis
done after the final such iteration. For each sample, the malfind output was manually investigated, to verify that the iodine output correctly emulated the discovered shellcode, correctly detected any injected PE files, and correctly skipped any zero-regions. Then, any functions which looked like they might be involved in an API hook were checked against the apihooks plugin, to see if this was the case. Once this verification was complete, the iodine output was compared against the ground-truth provided by the lab guide to verify that detected shellcode was indeed the result of code injection, and that processes determined to be hosting such code were indeed infected.

The Zeus sample was analyzed first, and its iodine output is shown in Figure 2.19. Indeed, each of the detected pieces of shellcode, denoted by the "Branch Addr" column, leaves its hosting VAD, and each of the processes with shellcode detected in them are processes Zeus has injected code into. Furthermore, analysis through apihooks shows that the function ZwCreateThread was in fact hooked, which is indicated through iodine displaying the jump into that function.

Next, the malicious PDF sample was examined, as shown in Figure 2.20. Similarly to the Zeus sample, each of the three infected processes are correctly identified, and both of the jumps into API functions are indeed jumps into hooked functions. Additionally, the calls to 0xb20ff0 and 0xb1d9d0 eventually lead to calls to functions apihooks identifies as being hooked, as shown in Figure 2.21.

The output from analyzing the TDSS infected sample is shown in Figure 2.22. As with the previous samples, infected processes and hooked functions are correctly identified. Parameters passed to the ZwCreateEvent function can also be seen in Figure 2.23, which can give investigators a better idea about the behavior of the injected code. Parameters to functions called are typically only printed when they are deemed potentially interesting enough, typically the case with strings or flags, to avoid cluttering the output in a distracting way. As such, the parameters to RtlAllocateHeap are omitted, while the parameters to ZwCreateEvent are displayed.
<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
<th>Start Addr</th>
<th>Branch Addr</th>
<th>Branch Dest</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1352</td>
<td>wsctf.exe</td>
<td>9a0000</td>
<td></td>
<td></td>
<td>0x9a0000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>1540</td>
<td>explorer.exe</td>
<td>1a7000</td>
<td></td>
<td></td>
<td>0x1a7000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>1540</td>
<td>explorer.exe</td>
<td>1c5000</td>
<td></td>
<td></td>
<td>0x1c5000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>896</td>
<td>rund1132.exe</td>
<td>a9000</td>
<td></td>
<td></td>
<td>0xa9000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>896</td>
<td>rund1132.exe</td>
<td>d3000</td>
<td></td>
<td></td>
<td>0xd3000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>896</td>
<td>rund1132.exe</td>
<td>217000</td>
<td></td>
<td></td>
<td>0x217000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>896</td>
<td>rund1132.exe</td>
<td>220000</td>
<td></td>
<td></td>
<td>0x220000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>30c0000</td>
<td></td>
<td></td>
<td>0x30c000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>2d830000</td>
<td></td>
<td></td>
<td>0x2d83000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>596d0000</td>
<td></td>
<td></td>
<td>0x596d000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>36590000</td>
<td></td>
<td></td>
<td>0x3659000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>3bb30000</td>
<td></td>
<td></td>
<td>0x3bb3000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>5fc80000</td>
<td></td>
<td></td>
<td>0x5fc8000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>544</td>
<td>winlogon.exe</td>
<td>75c80000</td>
<td></td>
<td></td>
<td>0x75c8000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>2488</td>
<td>TPAutoConnect.e</td>
<td>b8000</td>
<td></td>
<td></td>
<td>0xb8000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>2488</td>
<td>TPAutoConnect.e</td>
<td>d50000</td>
<td></td>
<td></td>
<td>0xd50000 0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>2928</td>
<td>wpa.exe</td>
<td>900000</td>
<td></td>
<td></td>
<td>0x90000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
<tr>
<td>2928</td>
<td>wpa.exe</td>
<td>aa0000</td>
<td></td>
<td></td>
<td>0xaa000  0x7c90d7d7 jmp ZwCreateThread + 0x5</td>
</tr>
</tbody>
</table>

**Figure 2.19** Output of *iodine* against a Zeus-infected image. The three rightmost columns have been cropped to save space.
<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
<th>Start Addr</th>
<th>Branch Addr</th>
<th>Branch Dest</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>644</td>
<td>csrss.exe</td>
<td>7f6f0000</td>
<td></td>
<td></td>
<td>lcall [edi]</td>
</tr>
<tr>
<td>1904</td>
<td>mscorsvw.exe</td>
<td>bc0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1764</td>
<td>IEXPLORE.EXE</td>
<td>5ff0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>firefox.exe</td>
<td>7d0000</td>
<td>0x5ff000a</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x7d0005</td>
<td>0x7c9163c8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jmp LdrLoadDll + 0x5</td>
</tr>
<tr>
<td>180</td>
<td>firefox.exe</td>
<td>25f0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>firefox.exe</td>
<td>36e0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>3530000</td>
<td>0x353074a</td>
<td>0xbb20ff0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call 0xbb20ff0</td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>2310000</td>
<td>0x3500ecf</td>
<td>0xbb20ff0</td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>28a0000</td>
<td>0x350000</td>
<td>0xbb20ff0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x3510000</td>
<td>0xbbd9d0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call 0xbbd9d0</td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>36a0000</td>
<td>0x36a0008</td>
<td>0x368f00e</td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>5500000</td>
<td>0x55000006</td>
<td>0xaccf7ae7</td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>92f0000</td>
<td>0x92f000b</td>
<td>0x7e42c4a3</td>
<td></td>
</tr>
<tr>
<td>3832</td>
<td>thunderbird.exe</td>
<td>ab0000</td>
<td>0xab1b72</td>
<td>0x8bb5ddda</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.20** Output of *iodine* against a sample infected by a malicious PDF. The three rightmost columns have been cropped to save space.

```assembly
call 0xbb20ff0
?js_GetClassPrototype...JS@@PAUClass@jss@@Z a5eb0

call 0xbb20ff0
?js_GetClassPrototype...JS@@PAUClass@jss@@Z a5eb0

call 0xbbd9d0
JS_SetGCPParameterForThread
malloc
b73b80
10004b5e
```

**Figure 2.21** Output of *iodine* against a sample infected by a malicious PDF. The three leftmost columns and several rows have been cropped to save space. The functions starting with "JS" or "?js" are detected as hooked by *apihooks*
The Windows 8 machine infected with mimikatz was perhaps even more illustrative to the investigator. Figure 2.24 shows the function call information yielded by the infected process. The shellcode can be seen collecting information about the context it is executing in by grabbing the module handle, header information, and then checking the environment for certain strings. As in the previous examples, injected shellcode was discovered and emulated, and hooked API functions were revealed in the process.

The three other machines infected with mimikatz (or, malware which included mimikatz as its payload) were somewhat more interesting. Figure 2.25 shows the sole potential shellcode region for the Windows Server image, out of almost a hundred processes found by malfind. It is unclear immediately what is going on, as the functions called (aside from memset) are fairly arcane, and the jump was done indirectly. It seems possible that iodine might have picked up some junk code. Investigation with volshell, however, quickly indi-
Figure 2.25 Shellcode discovered in Mimikatz infected Windows Server machine.

Figure 2.26 Shellcode discovered in Mimikatz infected Windows Server machine, examined with volshell.

cated that this is likely a forensically interesting region. Figure 2.26 shows the disassembly at the beginning of the region malfind detected. An address is moved into rax before the jump, so the address itself was examined next, as shown in Figure 2.27. The value stored at 0x7eff19a138 does indeed appear to be another address, and so that address was again disassembled, as shown in Figure 2.28. This seems to be a stub of shellcode setting up a function call, calling that function, and then returning. Analyzing the next of these infected machines found a very similar injection, as shown in figures 2.29 and 2.30. The output has been broken down again for readability, but the former shows again the same function call pattern as seen in the server, after another jump to [rax] 10 bytes into a given memory region. The latter shows a Firefox process exhibiting similar environment-querying behavior as the Mimikatz infected Windows 8 instance, shown in Figure 2.24, as well as a hook installed in the same function. The last of these machines seems to have captured Mimikatz at its most active. Figures 2.31 and 2.32 show some of the injected processes discovered in the final sample. Omitted is a Firefox injection identical to 2.30.
In [5]: \text{dis(0x7fefefed0a0)}

\begin{verbatim}
0x7fefefed0a0 48894c2408 MOV [RSP+0x8], RCX
0x7fefefed0a5 4889542410 MOV [RSP+0x10], RDX
0x7fefefed0aa 4c89442418 MOV [RSP+0x18], R8
0x7fefefed0af 4c894c2420 MOV [RSP+0x20], R9
0x7fefefed0b4 4883ec48 SUB RSP, 0x48
0x7fefefed0b8 660fd6442420 MOVO [RSP+0x20], XMM0
0x7fefefed0be 660fd64c2428 MOVO [RSP+0x28], XMM1
0x7fefefed0c4 660fd6542430 MOVO [RSP+0x30], XMM2
0x7fefefed0ca 660fd65c2438 MOVO [RSP+0x38], XMM3
0x7fefefed0d0 488d4c2450 LEA RCX, [RSP+0x50]
0x7fefefed0d5 488d542420 LEA RDX, [RSP+0x20]
0x7fefefed0da 458bc2 MOV R8D, R10D
0x7fefefed0dd e8be4e1400 CALL 0x7feff131fa0
0x7fefefed0e2 4883c448 ADD RSP, 0x48
0x7fefefed0e6 c3 RET
\end{verbatim}

\textbf{Figure 2.28} Disassembling the address found in Figure 2.27

\begin{verbatim}
2072 explorer.exe 407e0000 0x4030010 0x0 jmp qword ptr [rax]
\end{verbatim}

\textbf{Figure 2.29} Infected explorer.exe process.

\begin{verbatim}
6646 firefox.exe 200000 0x20003 0x1ff90 ja 0x1ff90
\end{verbatim}

\textbf{Figure 2.30} Infected Firefox process.
Figure 2.31 Hooked LoadLibrary calls, followed by calls to undocumented Windows functions, and another injection into explorer.exe.

The last of this set of memory images examined was infected by Stuxnet. Figures 2.33 and 2.34 show the most interesting pieces of the output. Notably, all of the infected regions make similar calls to `GetProcAddress` and `VirtualProtect`. As shown in the latter figure, twelve such calls are made, corresponding to fetching the address of six separate functions and then changing their permissions to `0x80`, or `PAGE_EXECUTE_WRITECOPY`, which is exactly what is needed to hook these functions. Indeed, in Symantec’s Stuxnet Dossier [40], it is noted that stuxnet hooks six functions in Ntdll, `ZwMapViewOfSection`, `ZwCreateSection`, `ZwOpenFile`, `ZwCloseFile`, `ZwQueryAttributesFile`, and `ZwQuerySection`, and that this pattern of hooks is an indicator of compromise for Stuxnet. In this memory sample, these six functions are found at the addresses fetched in Figure 2.34.

One final sample was examined, outside of the set provided through the Volatility training course. The SANS institute issued a challenge for detecting and studying the Skeleton Key [57] attack, and provided an infected memory image for this purpose. This image was infected with Mimikatz, which includes an implementation of this attack. Figure 2.35 shows the
Figure 2.32 Code injection into Internet Explorer processes, and indication of many function hooks.
<table>
<thead>
<tr>
<th>PID</th>
<th>Name</th>
<th>Start Addr</th>
<th>Branch Addr</th>
<th>Branch Dest</th>
<th>Instruction</th>
<th>Function Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>668</td>
<td>services.exe</td>
<td>7F6F0000</td>
<td>940000</td>
<td></td>
<td>call GetProcAddress</td>
<td>GetProcAddress VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x94003C0</td>
<td></td>
<td>call VirtualProtect</td>
<td>GetProcAddress VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x94003D0</td>
<td></td>
<td></td>
<td>GetProcAddress VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x7C8001A0</td>
<td></td>
<td>call GetProcAddress</td>
<td>GetProcAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0x7C8001A0</td>
<td></td>
<td>call VirtualProtect</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>668</td>
<td>services.exe</td>
<td>13F0000</td>
<td>B70000</td>
<td></td>
<td>call GetProcAddress</td>
<td>GetProcAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xBF0000</td>
<td></td>
<td>call VirtualProtect</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xBF0000</td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>940</td>
<td>svchost.exe</td>
<td>D00000</td>
<td></td>
<td></td>
<td></td>
<td>GetProcAddress</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call Virtual Protect</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>1190</td>
<td>explorer.exe</td>
<td></td>
<td>255000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>868</td>
<td>lsass.exe</td>
<td></td>
<td>80000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>868</td>
<td>lsass.exe</td>
<td></td>
<td>100000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>1920</td>
<td>lsass.exe</td>
<td></td>
<td>0F0000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>1920</td>
<td>\lsass.exe</td>
<td></td>
<td>000000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
<tr>
<td>1920</td>
<td>\lsass.exe</td>
<td></td>
<td>870000</td>
<td></td>
<td>GetProcAddress</td>
<td>VirtualProtect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VirtualProtect</td>
</tr>
</tbody>
</table>

**Figure 2.33** Injected regions discovered in several process of a Stuxnet infected sample.

**Figure 2.34** Function call list produced by code injected into lsass.exe.
cropped output of \textit{iodine} on this sample, specifically the functions called during emulation and their addresses. Figure 2.36 shows the Mimikatz source code of exactly the function being emulated. Specifically the function calls we see are found on lines 6, 8, 11, 12, 15, 16, and 19. Clearly the addresses shown in this code are not the addresses shown in Figure 2.35. This is a feature of Mimikatz. The strings which are repeated values such as $4a$ are searched for in process memory and overwritten with the correct addresses. The functions that overwrite which values can be seen in Figure 2.37.

Importantly, the \textit{iodine} plugin in every case correctly detected injected shellcode, ana-
Figure 2.37 Mimikatz source code for overwriting patterns in memory, truncated for clarity.

analyzed the shellcode, and presented the results in a concise, readable manner, without any analysis required by a potential investigator aside from running `malfind` on a sample to produce the requisite JSON file. While the process to clean up the results was iterative, so that functions could be implemented in HookTracer, this is not a necessary step and the results are correct without it. As the plugin is more widely used, and more functions are implemented, this will become less of an issue. That being said, implementing a new function is usually a matter of ten to thirty minutes of work, as HookTracer has built in functionality to simplify the process.

2.5 Related Work

As mentioned in Chapter 1, emulation is commonly used in computer security for malware analysis [136, 62, 12, 68] and has been for many years, but until recently all of this work required a whole-system emulator such as QEMU or Bochs [71]. Likewise, significant work on virtual machine introspection (VMI) [72, 2, 3, 4, 60, 46, 96, 38] in relation to malware analysis has been done in the past decade. VMI allows processes inside of a guest machine to be monitored by the host, granting relative safety from the malware. Finally, sandboxes such as Cuckoo [33] allow for automated running of and report generation about malware.
While such technologies are useful for analyzing malware, they are unfortunately not suitable for memory analysis because of the substantial differences between an executable on disk and one loaded into memory. Because of this, executables cannot usually be carved out of memory and run directly. Additionally, memory-only malware does not share most of the structure found in an executable, and thus cannot be bundled into an executable and run directly. These systems also require extensive instrumentation of the target machine and specialized labs that cannot be reasonably used in incident response situations.

Emulation, and particularly unicorn have been applied to memory forensics as well, with two recent efforts. The first, ROPMEMU [48, 123], automatically detects ROP chains [78] in memory. The second detects the "Gargoyle" attack, where a malware hides itself by changing permissions on its code using a timer. Neither of these projects overlap with HookTracer, however they are still important to note.

Effort has been made to analyze API hooks in memory through a research project and the resulting Volatility plugin, apihooksdeep [122]. This plugin uses fuzzy hashing [65] to generate a hash set of the hook code. First, the owning module must be extracted from memory, and then targeted by a tool that actually generates the hash set. The hash set can be used as a filter to exclude known-good hooks from being outputted. This process is fairly inaccessible, since generating a reasonable set of hashes requires a clean memory image containing every process from which the investigator wishes to exclude running and hooking code. Additionally, these hashes may need to be updated when updates are pushed to the permitted applications.

Lastly, a very recent (awaiting publication at the time of this writing) effort has been made to automatically determine capabilities of malware found in memory images [5]. Similar to the work presented here, FORECAST analyzes and emulates malicious code found in a memory image, with a specific focus on the API functions called. However, FORECAST uses this information to make predictions about the high-level capabilities of the malware found in an infected system, whereas this work is focused on helping investigators determine if
infection is present on the system.
Chapter Three. Seance

Memory forensic tools such as Volatility and Recall [128, 47] rely heavily on being able to reproduce the functionality of code on a system under analysis. In particular, memory forensic tools need to access certain members of data structures, typically in the same way related code does. As such, changes to critical data structures have the potential to badly break tools, with identification and correction of the underlying problem requiring significant effort. Such breaks could manifest as anything from the tool not running to incorrect results, with no appropriate error messages. An analyst in the middle of an investigation, then, may have to wait on an official fix to the tool to be issued, or come up with one themselves. In either case, such a delay could range from stressful to unacceptable. In order to address this problem, a binary analysis tool called Seance was developed to automatically detect changes to data structures based on changes to code that natively interacts with those structures.

This effort attempts to leverage binary analysis capabilities to automatically verify compatibility of forensic tools with new versions of forensic targets. Take for example an update to the some language’s runtime system. An investigator might be using memory analysis tools reliant on knowing the location of certain members of various data structures used by the system. Determining if her tool still works might require a significant reverse engineering effort to see what, if anything, has changed in a multitude of different structures or functions that access them. Without such an effort, the investigator would not know if her tool would produce nonsense results, or even misleadingly incorrect results, either of which are unacceptable in a forensic context. Seance alleviates much of this effort. Initially, work was done to try and use HookTracer for this purpose, but fundamental design choices in HookTracer meant that it was not suitable in this domain. Namely, during emulation, only a single path is taken through the target code since all values are predetermined. This results in situations where, for instance, an access to a data structure that triggers on a specific set of circumstances is completely missed. To address this problem, a new backing API, which
powers Seance, was designed as an alternative to HookTracer, and which takes inspiration from the design and motivation of HookTracer to instrument execution of code for forensic ends. Instead of making use of unicorn directly, this API is built on top of angr (see Section 1.4.2) to take advantage of its symbolic execution (see Section 1.4) capabilities. The goal behind this API is to get more detailed execution information than HookTracer can provide, and to that end results in two major improvements:

- **Control flow graph (CFG) generation**: If one were to construct a control flow graph for emulation conducted with either of the HookTracer plugins, the result would be a single path, possibly with cycles arising from loops. This is because each time the emulation encounters a branch it only gathers information about one tine of the fork. Instead, using Seance’s API, both paths are taken, and information about program state for each execution is recorded. Using some of angr’s built in analyses, a full control flow graph can be constructed for each hook.

- **Memory Access Information**: Whenever the emulated code makes an attempt to read or write memory, the access is recorded and associated with the particular state where it happened. Due to the symbolic nature of some values during emulation, these accesses may be referencing addresses or writing values that have no concrete value. Once all of the emulation is completed, however, for each ending state reached, every memory access that occurred over the path leading to that state is concretized by the path-specific constraints at the time of the access. The end result of this is a record of all memory accesses conforming to the context in which they were created. Take for example code that reads in a symbolic value, say $V_x$, and branches based on if $V_x$ is less than 10 or not. After this, the code writes the value of $V_x$ to memory. In this analysis, then, the result would be two records of the write, one with some concrete value less than 10 and one with some concrete value greater than or equal to 10.

The first application of this API is towards the problem detailed above: Seance can be
deployed targeting functions which access forensically important members of relevant structures and return a listing of the offsets into the target data structure where were accessed. This can reveal if new members have been added to that structure, if members have been removed, if types have changed, or if the function accesses a completely different structure. Additionally, the CFG capabilities of Seance can be used to see changes to the function itself that may impact the correctness of tools or reveal new implementation details.

In this chapter, examples of code are given which include various forensically valuable data structures, but that change significantly across versions. Additionally, it is demonstrated that Seance detects all such significant changes through the creation of a database containing automatically generated fingerprints of the target data structures across a test set of code versions, generation of signatures from several temporally discontiguous versions of the same code, and finally a comparison of the new results to the database. Furthermore, it is shown that this process indeed provides valuable information about the nature of those changes, and thus dramatically reduces the burden they impose on investigators and maintainers of forensic tools.

3.1 On Reading Bones

This section documents the current approaches to data structure layout reconstruction used in memory analysis frameworks, along with the limitations to these approaches. This topic will be partially discussed in Section 3.4, but given its complexity and central role in motivating our research effort, it has also been included this section for completeness.

When a memory forensic developer wishes to learn the layout of a data structure of interest, there are three possibilities [75]. The first two, source code review and use of debugging symbols, are generally easier than the third, binary analysis, but are not always accurate, complete, or even feasible. In those situations, binary analysis is the only choice available. Luckily, if an application accesses target members of a data structure, those
accesses will be encoded within instructions in the executable module. This means binary analysis is not only possible in virtually every context, but it is also guaranteed to encode the correct offset.

3.1.1 Source Code Review

Approach

This method uses access to the source code of an application to determine the name, layout, and purpose of data structures of interest. By knowing the compiler rules for how data structures are laid out in memory compared to their source code representation, it is often possible to manually build out the data structure layout.

Limitations

The first limitation of this approach is its extremely time-consuming and error prone nature in cases of large code bases. One miscalculation for a structure member will break the derived offsets for all remaining members. Considering dozens of data structures are often needed for deep analysis and that each data structure often has dozens of members, the chance of such miscalculation occurring is high. Furthermore, to support the 32-bit and 64-bit versions of applications, this process must be repeated twice for each module release. Also complicating matters is that the compiler and linker optimizations can create significant changes in the data structures and ultimately the final view of the binary. As discussed in related works [18, 102, 29], compiler alignment of structure members has a drastic effect on this layout and the alignment choices are not always discernible just from viewing the source code.
3.1.2 Use of Debugging Symbols

Approach

When debug files contain complete type and symbol information then nothing else must be done except to convert that information into a representation that the memory analysis framework understands. The two main file formats for this information are PDB files from Visual Studio and DWARF files on Linux and Mac. The PDB file format has been reverse engineered to enable extracting all information and the DWARF format is fully documented.

Limitations

The main limitation of this approach is that complete debugging information is not published for many of the modules needed for memory analysis. As mentioned previously, some vendors publish debug information of their production executable modules, and the two most commonly used in memory forensics are the Microsoft symbol server and the kernel build repositories maintained by various Linux distributions. Unfortunately, those published by Microsoft are usually only complete for the kernel executable itself.

In the situation where source code of a target module is available but the vendor does not publish debug files, researchers will often compile the source code on their own with debugging enabled to generate a local debug information file. When the compiler version and settings of the target module are known then this workflow is generally accurate, but accuracy decreases when the exact build environment configuration cannot be replicated. Furthermore, there are platforms, such as macOS, that mix closed and open source code, so compiling the open source code is not always possible with the absence of code from closed source components.
3.1.3 Binary Analysis

Approach

Given the inability of the previous methods to work in many situations, memory analysis capabilities are often built using binary analysis. This approach requires an expert investigator to find the functions within a module that reference needed structure members.

Limitations

This is a time consuming approach that requires an expert to manually reverse engineer many components of target modules. The expert must then manually record all offsets and types discovered into the target framework’s format. This makes the effort not only extremely time consuming, but also error-prone and non-repeatable.

Shared Limitations

There is also a significant, shared limitation of all three approaches in that they only attempt to acquire the offsets of needed members of a data structure. As demonstrated with Objective-C in Section 3.3.1, when attempting to correctly support a wide variety of versions, just having the offset is not enough. Instead, the semantics of the operations performed on that offset must be understood, so that frameworks can find and report valid artifacts. The combined issues and limitations of current approaches all necessitate the need for automated approaches to processing modules at the instruction level.

3.2 How to Commune With Spirits

3.2.1 Lighting The Candles

This section describes the implementation of Seance. Like HookTracer, a large part of Seance is implemented as an API that different programs can make use of. This engine is composed
of a few callbacks and several functions to aid in analysis.

The API assumes that the tools making use of it have already instantiated an angr project because turning such instantiation into an API call would require roughly the same amount of work for the person building the tool. An angr project is the basic structure that angr operates on, and is constructed by giving angr a binary to look at (this can also be a "blob," a binary with no executable structure such as a chunk of memory pulled from an image), and several options on how to interpret the binary. Most relevant to this work are "backend," "arch," and "segments," which give the structure of the binary (for instance PE, ELF, or blob), the processor architecture to use, and an array of segments. This last option is used in the case that the binary has sections that need to be treated as if they were at a different address, as is the case when trying to load a process’s whole address space as a single binary while maintaining a relatively manageable file size. Additionally, it is assumed that the tool using Seance starts the emulation itself with the callbacks it wants to use registered, but the callbacks themselves are provided by Seance. The four currently supported callbacks trigger on register reads and writes, and memory reads and writes. The behavior of these callbacks is as follows:

- **Emulation check:** The actual first check of these callbacks is of a global flag which is True if emulation is ongoing, or False otherwise. This is because post-emulation accesses during analysis still trigger the callbacks and should be ignored.

- **Check if the target can be found:** The first real step in each of these callbacks is to make sure the access targets a concrete register or memory address. If it does not, the callback exits after printing a warning. If it can, the specific register or address are recorded, along with the length of the access, in a globally stored record of reads or writes, depending on the access type.

- **Record data:** Next, in the case of a register write, a check is made that the register is not an instruction pointer as this would be called after every instruction. In all other
cases, or if this check comes back negative, the callback records the expression being written or read, the condition that triggered the access, the length of the access, and finally an ordered list of basic blocks visited.

Seance also provides functionality to handle stashes. Stashes are groups of states created during emulation, and all of the states in all of the stashes is what allows Seance to sort out what actions happened when, and with which results. This is done via the `handle_stashes` function, which iterates over all of the ending states. During each iteration, a list of basic blocks visited is collected and compared against the basic blocks recorded in each memory access. If the list of blocks in the access is completely contained in the history of the end state, the access is recorded as having taken place sometime during the execution that lead to this end state. For each of these accesses, the access address or register is concretized, checked against the bounds of the address space, and if it is in bounds, concretized and recorded as a string, a hexadecimal number, and as bytes. Additionally, each basic block visited is added to a list of permissible blocks to visit during CFG generation. A record of each end state along with their corresponding accesses is created, and returned along with the list of permissible blocks.

The next important functionality provided by the API is generation of the CFG. The function `generate_cfg` handles this, and expects a start address, the current project, and a permit list of blocks. `generate_cfg` attempts to construct the CFG, and if it exists, prints out the number of nodes and edges, and returns the graph. From here, if the user wants, she can call the function `print_cfg` to output the given CFG as a png.

In addition to providing analysis capabilities, Seance also supports scanning of the target binary to gather information to help the analysis. If a user has a specific function she wants to target, in terms of a symbol name, Seance’s `get_symbol_addr` function will attempt to locate that symbol’s starting and ending addresses. To find the start address, `angr` is first leveraged to see if the symbol is included in the information packaged with the binary. If not,
Seance checks for a PDB file, if one exists, and tries to resolve the symbol address in that. In either case, if the symbol is located, Seance then attempts to locate an ending address for the target function by enumerating the symbols and finding the closest symbol which begins at a larger address than the target. Then, using angr to process the instructions, Seance scans backwards from that address looking for the first instruction that is not a \texttt{nop} or \texttt{int 3} instruction, both of which are common ways to fill space between functions. Once Seance finds such an instruction, the address of that instruction is returned as a potential ending address of the symbol. Further, instead of just targeting a specific symbol, a user may want to target a specific portion of that symbol. This might be the case if some function is not passed a data structure as a parameter, but instead calls a separate function which returns that structure, and then the outer function proceeds to operate on the structure. In such a case, the user does not need to know the whole behavior of the symbol they choose to target, but just the behavior between calls to one or more other functions. To support this use case, Seance provides the \texttt{find_bb_containing} function, which accepts a list of one or more function names, and scans the binary for contiguous sequences of basic blocks which each call one of those functions, and returns a list of starting addresses for such chains of basic blocks. The user can then cross reference the addresses of those basic blocks with the addresses inside their target (outer) function. In cases where the outer function is broken into non-contiguous memory regions, as is sometimes the case, the \texttt{find_jumps_out} function will scan the target function for jumps, either \texttt{jmp} or \texttt{jcc} instructions, which end outside of its starting memory region, as found by the methods mentioned above. Then, the \texttt{get_jump_regions} breaks these jump addresses into regions where no jump target is more than 0x200 bytes from the next target, which are then included when cross-referencing the starting addresses of basic block chains. This number was chosen after examining one such function, found in tcpip.sys, and seeing how much distance there was between pieces of the function. The chosen number works well in practice, but can be tweaked if needed.

Lastly, Seance provides functionality for the processing of data produced during the
instrumented symbolic execution. The function \texttt{sort_access} consumes the read and write data from the execution and yields:

- Offsets from initial register values, for each register
- Offsets from pointers, for each pointer
- Sources for the pointers discovered, to two degrees of indirection

An example of the result of this post-processing can be seen in Figure 3.1. This post-processing is what allows an investigator to clearly see accesses to data structure members, and allows for the construction of databases of access-patterns for different versions of a binary.

### 3.2.2 Parting the Veil

This section discusses how the components described in Section 3.2 were composed to create Seance. It is important to first cover exactly what the inputs and outputs of Seance are. To this end, Seance consumes:

- **A binary** Despite research efforts, binary analysis remains impossible without a binary.
- **A target** The user must provide a target function she wishes to analyse. This can be either the name of a symbol for Seance to resolve, or a starting and ending address. The address mode was created to assist in debugging, but was left in as uses cases for it might manifest in the future.
- **Optional list of sub-targets** If the user wants to single out a specific section of her target function (as described in 3.2) in which one or more functions are called, she may specify a list of sub-targets, each of which may be either a symbol name or an address.
- **Optional PDB data file** In the case that the binary to be analyzed has an associated PDB file, Seance will try and find a file containing information derived from this file.
Figure 3.1 Reading memory accesses back out of JSON files generated from analysis of a function in Windows' tcpip.sys.

through use of the pdbpares tool [36] as described in Section ??.
Specifically the script pdb_print_gvars.py, provided by pdbparse, must be called on the associated PDB file. The output of this should be redirected to a file named "gvars" in the directory which contains the binary under analysis. Seance will look for this gvars file, which contains symbols and their offsets into the binary, and use this to resolve symbols when angr cannot.

• User Options Lastly, the user may provide command line options to further influence
the behavior of Seance. The most important of these are specific registers to target, whether or not to zero-fill registers or memory, and whether or not to skip function calls during symbolic execution.

And then from these inputs, Seance yields:

- **A printed canonical CFG** The first analysis Seance does is production of a CFG of the targeted code. A representation of this CFG is emitted as a PNG file.

- **Analysis of the CFG** The CFG produced is analyzed, yielding the number of nodes, number of ending states, and number of function calls, which are saved in a JSON file.

- **Raw information from symbolic execution** For debugging purposes, Seance saves the raw data produced from the instrumented symbolic execution. This is the same data Seance will further process to produce the following items, and as such represents something of an intermediate state for Seance.

- **Information for each ending state** Symbolic execution may result in several different ending states for a given starting point. From the raw information noted above, Seance pulls apart the specific information from each ending state, produces a CFG for that particular state’s history, prints it and saves the execution information. Additionally, Seance produces the offset and pointer information for each ending state and saves these as well. This is all again mostly for debugging purposes, as the information for a single ending state is often easier to examine than that for the whole run.

- **Offset information** Seance further processes the offset information, that is the information about memory accesses to addresses at an offset from the initial value in each register, produced for each state. This information is consolidated, repetitions are removed, and a JSON file is emitted with this information. For the sake of any humans involved, Seance also emits a pretty-printed version of this data.
• **Pointer information** As above, but for information about accesses to memory addresses at an offset to another memory access. Again, a JSON version and a human-readable version of this information are produced.

• **Back-traced pointer information** Seance tries to trace back any discovered pointers to their origin, two times over. Some data structures encountered were indeed substructures of another structure, and were accessed as offsets from pointers to pointers. To capture this behavior, Seance looks at any pointers discovered and checks if that pointer was originally pointed to by another pointer or register. This process is then repeated to capture second-degree indirection, and the results are combined and saved again to JSON and human-readable files.

The data emitted by Seance are further processed later as part of an automated workflow, but these are the basic inputs and outputs to the system. With this in mind, the actual computation of Seance is as follows:

1. **Parse inputs**

2. **Locate symbol** Find both the starting and ending address of a symbol, as described in Section 3.2.

3. **Generate canonical CFG** Generate the CFG through static means.

4. **Analyze canonical CFG** Collect the number of nodes, deadends, and functions called. Generate a list of deadends’ addresses.

5. **Symbolic execution**

   (a) **Install callbacks** Callbacks discussed in Section 3.2.

   (b) **Begin execution** Symbolic execution stops on finding a deadend in the CFG. Repeat for each deadend discovered in step 4.
(c) **Log memory accesses** Done automatically by the callbacks as execution happens.

6. **Pre-process execution data**

   (a) **Separate memory reads from writes** It is useful to know what type of access is being handled.

   (b) **Concretize any symbolic values** If the destination of an access is symbolic, concretize it so that it can be reasoned about more easily.

   (c) **Record concretized accesses**

   (d) **Generate permit-list of basic blocks** To check that execution behaved reasonably, generate a permit-list of basic blocks accessed during execution, to use in step 8.

7. **Process memory accesses** The mechanisms for this are described in Section 3.2.

   (a) **Collect offsets from registers**

   (b) **Collect offsets from pointers**

   (c) **Find pointer sources**

8. **Generate real CFG** Generate a second CFG, restricted to only touch basic blocks in the permit list from step 6d.

Now that the operational details of Seance have been covered, the next section will describe the post-processing of this output to integrate into an automated workflow.

### 3.2.3 Secrets from Beyond the Grave

The direct output of a particular run of Seance may be interesting, or even useful, in its own right, but cross-referencing offsets and CFG data from multiple runs quickly becomes
unmanageable. To further alleviate the burden on investigators, additional post-processing capabilities were developed to cross-reference multiple runs of Seance and produce concise, human-readable output. The most common reason to cross-reference multiple Seance runs is to compare runs targeting the same symbol, in the same binary, across multiple versions. As such, the post-processing iterates over each such run, collecting the JSON files containing offset and CFG information and adding them into a larger JSON file, tagged with the binary version. This is the database for this particular symbol.

Once a database has been created for a symbol, an additional utility can be used to cross-reference the output of that run against the database. For each binary version in the database, the following process is performed. First, the CFG is compared to determine if it exactly matches the current run, meaning if the number of nodes and deadends are the same, as well as the number of external functions called.

The offset-accesses are compared multiple ways, to gauge levels of similarity. First, a literal comparison is performed, checking if the offsets accessed are exactly the same, adding the binary version to a list on a match. If the match is not identical, that is if each offset in the new version does not have exactly the same associated list of offsets as in the old version, then a more fine-grained comparison is performed. In this case, duplicate offsets from each register and pointer are removed in the old and new versions, and another literal comparison is performed, adding the old version to a new (distinct from the "identical match" list) list if it matches.

If this more focused comparison does not match, the stripped offset lists are then compared on an access-by-access basis, checking if any of the accesses had the same offsets referenced between versions. If any of the accesses match here, the list of matching accesses is added to a third list along with the binary version.

The direct comparison of CFG data, along with three levels of access checks, leaves five outcomes:
1. Full fingerprint match, meaning the CFG and offset accesses were the same

2. Offset match with CFG change, meaning the offsets were the same but the code changed

3. Raw offset match, meaning the offsets were the same but their ordering and/or number of accesses changed

4. Specific access match, meaning only a subset of accesses in the list matched

5. No matches, meaning the CFG changed and no previously found offsets were matched

This presentation of results quickly tells investigators if their tools will work as intended or if a change is needed. It also informs developers as to which action(s) must be taken to support the new version.

Outcome 1) requires no changes to the framework, 2) and 5) indicate that the targeted function should be re-analyzed to ensure consistency of extracted artifacts, 3) generally occurs as a result of compiler code re-ordering between versions and does not require re-analyzing the function, and 4) requires re-analysis only if offsets relevant to the framework have changed.

3.2.4 Things Left Behind

The automatic binary analysis capabilities of Seance allow investigators to quickly and easily determine if a memory analysis framework supports a specific version of a particular binary. Seance’s analysis describe accesses to a particular offset or set of offsets which are not brittle to instruction changes, register changes, or even substantial compiler code re-ordering, as Seance’s analysis is driven at a higher level, through symbolic execution. Overall, these features will allow for a significant shift in the reliability and speed in which memory analysis frameworks can support a wide range of operating system and application versions. The next section details how all of these capabilities were leveraged to analyze forensically important binaries for both macOS and Windows.
3.3 Conversations with Dead People

3.3.1 Objective C Runtime

The first full evaluation of Seance was conducted against the Objective-C runtime. This was chosen as a target for several reasons, including that is commonly abused by malware on macOS systems, its sole previous research effort is largely outdated, and that it is open source.

In 2016, a paper was published at DFRWS that documented memory forensic algorithms for detecting the Crisis malware and its abuse of the Objective-C runtime [19]. This was accomplished through enumeration of Objective-C classes and data loaded into a process followed by checking for signs of malicious activity. This work only targeted macOS version 10.9, however, which stopped being supported in 2016 by Apple and is now six major OS versions of out date. Attempting to integrate this data structure and algorithm data into a new Volatility plugin to operate on the latest Objective C versions resulted in incorrect results.

The memory forensics community as a whole faces similar issues with applications and modules across a number of other operating systems. As manual compatibility checking is clearly unworkable in the face of the sheer scale of this problem, it was decided that an automated system for determining version compatibility of memory forensic algorithms was needed, and, hence, Seance was born.

Targeted Data Structures

The goal for revamping Objective-C analysis was to be able to gather all loaded classes in a process address space followed by the enumeration of instance variables and methods for each class. This information allows for the enumeration of active instances of each class and the decoding of variable values. Table 3.1 lists the structures and members needed to accomplish this, along with the functions that reference each member.
Table 3.1 Obj-C data structures targeted

<table>
<thead>
<tr>
<th>Member</th>
<th>Structure</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>i_var</td>
<td>getName</td>
</tr>
<tr>
<td>offset</td>
<td>i_var</td>
<td>getOffset</td>
</tr>
<tr>
<td>type</td>
<td>i_var</td>
<td>getTypeEncoding</td>
</tr>
<tr>
<td>imp</td>
<td>method_t</td>
<td>getImplementation</td>
</tr>
<tr>
<td>name</td>
<td>method_t</td>
<td>getName</td>
</tr>
<tr>
<td>count</td>
<td>NXHashTable</td>
<td>NXEmptyHashTable</td>
</tr>
<tr>
<td>buckets</td>
<td>NXHashTable</td>
<td>NXFreeHashTable</td>
</tr>
<tr>
<td>nbBuckets</td>
<td>NXHashTable</td>
<td>NXInitHashState</td>
</tr>
<tr>
<td>count</td>
<td>NXHashTable</td>
<td>NXResetHashTable</td>
</tr>
<tr>
<td>isa</td>
<td>objc_object</td>
<td>object_getClass</td>
</tr>
<tr>
<td>bits</td>
<td>objc_class</td>
<td>removeSubclass</td>
</tr>
</tbody>
</table>

Versions Tested

The testing was conducted using versions of the Objective-C runtime corresponding to macOS versions 10.11.0 through 10.15.6. To gather the versions needed for testing, libobjc.dylib was first extracted from each of these macOS versions. These files were then de-duplicated based on the SHA1 hash of the library and stored which Objective-C versions mapped to each hash. In total, this set was comprised of 21 different files.

Methodology

In an attempt to simulate a realistic investigative scenario in testing, a Seance database was constructed from a selection of fourteen of the files, as described in Section 3.2.4. To construct a database that might reflect the experiences of a seasoned investigator, more files were included in the database than not, and since investigators do not typically get
### Table 3.2 Part 1 of a long table of Objective C functions analysed

<table>
<thead>
<tr>
<th>Structure Function</th>
<th>Parameter Register</th>
<th>Exact Match</th>
<th>Offset Match</th>
<th>CFG Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>NXHashTable</td>
<td>NXHashTable *</td>
<td>10.14.0-10.15.6</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>NXEmptyHashTable</td>
<td>rdi</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>NXHashTable</td>
<td>NXHashTable *</td>
<td>10.13.0-10.14.3</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>NXInitHashState</td>
<td>rdi</td>
<td>10.14.4-10.14.6</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>NXHashTable</td>
<td>NXHashTable *</td>
<td>10.15.0-10.15.6</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>NXFreeHashTable</td>
<td>rdi</td>
<td>10.13.4-10.13.3</td>
<td>¬(10.13.0 – 10.13.3)</td>
<td>-</td>
</tr>
<tr>
<td>NXHashTable</td>
<td>NXHashTable *</td>
<td>10.14.4-10.14.6</td>
<td>¬(10.13.0 – 10.13.3)</td>
<td>-</td>
</tr>
<tr>
<td>NXResetHashTable</td>
<td>rdi</td>
<td>10.15.0-10.15.6</td>
<td>¬(10.13.0 – 10.13.3)</td>
<td>-</td>
</tr>
<tr>
<td>ivar</td>
<td>Ivar</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>getName</td>
<td>rdi</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>ivar</td>
<td>Ivar</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>getOffset</td>
<td>rdi</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>ivar</td>
<td>Ivar</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
<tr>
<td>getTypeEncoding</td>
<td>rdi</td>
<td>ALL</td>
<td>ALL</td>
<td>-</td>
</tr>
</tbody>
</table>

To choose which machines they analyze, random chance was used to select which files were included. Once the database was constructed, each of the files left out of the database, the experimental group, were compared against the database as described in Section 3.2.4 for each of the target functions.
### Table 3.3 Part 2 of a long table of Objective C functions analysed

<table>
<thead>
<tr>
<th>Structure</th>
<th>Parameter</th>
<th>Exact Match</th>
<th>Offset Match</th>
<th>CFG Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>Function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>method</td>
<td>Method</td>
<td>10.11.0-10.15.3</td>
<td>SAME</td>
<td>-</td>
</tr>
<tr>
<td>getImplementation</td>
<td>rdi</td>
<td>10.15.4-10.15.6</td>
<td>SAME</td>
<td>-</td>
</tr>
<tr>
<td>method</td>
<td>Method</td>
<td>10.11.0-10.15.3</td>
<td>SAME</td>
<td>-</td>
</tr>
<tr>
<td>getName</td>
<td>rdi</td>
<td>10.15.4-10.15.6</td>
<td>SAME</td>
<td>-</td>
</tr>
<tr>
<td>objc_object</td>
<td>id</td>
<td>10.12.0-10.14.6</td>
<td>SAME</td>
<td>ALL</td>
</tr>
<tr>
<td>getClass</td>
<td>rdi</td>
<td>10.15.0-10.15.6</td>
<td>SAME</td>
<td>ALL</td>
</tr>
<tr>
<td>objc_class</td>
<td>Class</td>
<td>10.13.4-10.13.6</td>
<td>10.14.4</td>
<td>-</td>
</tr>
<tr>
<td>removeSubclass</td>
<td>rdi</td>
<td>10.11.4-10.13.5, 10.14.5-10.15.0</td>
<td>10.15.1-10.15.6</td>
<td>-</td>
</tr>
<tr>
<td>objc_class</td>
<td>Class</td>
<td>10.13.4-10.13.6</td>
<td>10.14.4</td>
<td>-</td>
</tr>
<tr>
<td>removeSubclass</td>
<td>rsi</td>
<td>10.11.4-10.13.5, 10.14.5-10.15.0</td>
<td>10.12.6, 10.15.2, 10.15.6</td>
<td>-</td>
</tr>
</tbody>
</table>

#### Analysis Results

To analyse the results, the differences detected were first compared to the chronology of the files. As expected, files which Seance registered as identical matches or identical matches with a CFG change for a given function, formed contiguous groups across Objective-C versions.

For example, `NXFreeHashTable` generated exact matches on three distinct groups of files, 10.13.0 - 10.14.3, 10.14.4 - 10.14.6, and 10.15.0 - 10.15.6 whereas `NXEmptyHashTable` generated matches for only one large grouping, 10.14.0 - 10.15.6. In contrast, in the relaxed
mode where only the parameter of interest is matched, then **NXEmptyHashTable** matches on every file tested.

Similarly, **NXResetHashTable** matches usage of the first parameter for all versions except 10.13.0-10.13.3, which correspond to exactly one of our test files. **method_getName** does not generate any additional matches under these relaxed conditions, but even with exact match enabled, only has two variants. The first covers 10.11.0 - 10.15.3 and the second 10.15.4 - 10.15.6. Interestingly, only the **object_getClass** function actually changes its CFG without using the parameters differently. This highlights the importance of CFG matching, since an investigator must check that the data structure is still used in the same way under the new algorithm.

One particular function, **removeSubclass**, did not produce any exact matches, only parameter matches, but these, too, coincided with temporal boundaries. However, in this case, a pattern was not readily apparent just by looking at the results file, and required source code analysis to make sense of the differences. A more concise, and detailed, breakdown of the results is presented in tables 3.2 and 3.3.

This brings us to the second stage of our analysis, where the source code corresponding to each file was examined to verify the changes reported by Seance. In this stage it was discovered that in most cases when Seance detected changes in offset or CFG, there was no corresponding change to the function’s source code. The exceptions to this were the functions **object_getClass** and **removeSubclass**. In the source file **objc-object.h**, the inlined function **getIsa**, which **object_getClass** uses, was substantially changed between OSX versions 10.11.6 and 10.12.0, and again between 10.14.6 and 10.15.0. Similarly, in the source file **objc-runtime-new.mm**, the function **removeSubclass** underwent major revisions between OSX versions 10.14.3 and 10.14.4, and again between 10.14.6 and 10.15.0. Finally, to verify the temporal boundaries where there was no source code change, disassemblies of the functions **NXEmptyHashTable**, **NXFreeHashTable**, and **NXResetHashTable**, where Seance detected changes but source code analysis revealed nothing, were examined. In each of
these cases, semantically meaningful differences in the assembly instructions were discovered. These differences could be the result to changes in the header files, compiler options, or compilation tool chain used.

In studying the source code corresponding to material changes reported by Seance, it was found that the algorithm to collect the variables and methods belonging to a class had substantially changed in Objective-C for macOS 10.15.0. In particular, the bits member (accessed by the removeSubclass function), which previously pointed to a class_rw_t structure, could now point to either a class_rw_t or a class_ro_t depending on the class’ state. Furthermore, the class_rw_t structure itself had been broken into two separate structures. This discovery illustrates the need for data structure layout extraction processes to not only find the correct offset, but also to verify the operations performed on the offset. In this situation, the bits member is used across versions, but the type it references has changed in such a drastic manner that just knowing its offset is not enough to correctly recover artifacts. Seance pointed directly to this discrepancy.

Analysis Conclusions

The evaluation of Objective-C has demonstrated that Seance is capable of analyzing a wide variety of versions of a real-world library. Source code analysis was initially used to derive the function list for Seance to analyze to find members, while the rest of the process was automated. The results of this automated processing highlighted new structure offsets across versions, as well as where just updating a structure’s offset would not be enough to accurately support analysis. This evaluation also pointed out the pitfalls of source code-only review, including that substantial changes can happen in the compiled form, as well as the downsides of data structure layout extraction workflows that only examine changes in offsets.
3.3.2 Evaluation - Windows Networking Stack

Analysis of the data structures of the Windows networking stack provides extremely valuable artifacts during an investigation. Through this analysis, an investigator can uncover all listening sockets and connections, network interfaces in promiscuous mode, and can map network activity to the process responsible for it. These data structures also contain timestamps that indicate when specific activities started and/or ended.

As incident response investigations often start as a result of a network indicator, such as a system contacting a known-bad IP address or resolving a known-bad hostname, it is a significant advantage if the investigator can quickly determine which processes were responsible for the malicious behaviour. Importantly, uncovering the connection creation time allows including recovered connections into investigative timelines.

Unfortunately, history has shown that key parts of these data structures vary greatly between tcpip.sys versions and that frameworks have not kept up with the changes. Browsing the Volatility 2 and Rekall issue trackers finds over twenty tickets related to connections not being reported or key metadata, particularly a connection’s create time and owning process, not being reported correctly.

Knowing the importance of recovering network activity from memory samples, Seance was leveraged to widen the support of tcpip.sys versions in Volatility.

Targeted Data Structures

For this evaluation, the TCP_ENDPOINT data structure was targeted. This structure holds the creation time, owning process, state, address family, and local and remote IP address and port for each connection. Figure 3.2 displays this structure and its related-structures as defined by Volatility for the base version of Windows 10. As illustrated in the figure, TCP_ENDPOINT holds the state, port information, owner, and create time information directly in the structure. Recovery of the address family and remote and local IP addresses
requires use of the related structures.

Figure 3.2 Data structures for recovering network connections

To support analysis with Seance, functions inside of tcpip.sys that accessed these members needed to be determined. To start this process, a ground truth was generated of the TCP_ENDPOINT structure layout for particular versions. To accomplish this, a two-step approach was taken. First, the source code of Volatility 2 and, later, Volatility 3, was examined to determine the versions for which they had TCP_ENDPOINT defined. For Volatility 2, analysis showed that the entire structure was defined for the base versions of Windows 10 and that the Owner field was updated for version 15063. No other version had updated offsets for any members.

When analyzing Volatility 3, it was initially found that there was no support. Later, though, it was discovered that there was a very recent effort, beginning in December 2020, to bring support to Volatility 3. From a review of this effort, the structures appear accurate from version 15063 through version 19041, but there is no support for 20H2 or for the pre-2017
Windows 10 versions. It was also noted that the JSON files containing the structure layouts include a comment that they were created by hand. Furthermore, a few related commits indicating the structure layouts were incorrect were discovered. This is not surprising, as manual creation of complicated structure layouts is currently a very difficult, labor-intensive task. Eliminating this manual, error-prone process was a key motivating factor for this research.

Once the offsets used by Volatility 2 and 3 were obtained, an intensive binary analysis effort against tcpip.sys was undertaken to discover functions that reference the needed structure offsets. Given the complexity of tcpip.sys, this was a non-trival task. Analysis initially revealed that the State member is referenced in TcpCanTcbSend, but later testing with Seance revealed that a different function was needed to support Windows 10 version 20H2. This is discussed further in Section 3.3.2. It was also determined that the Owner and CreateTime fields are accessed in TcpCreateAndConnectTcb, and that all remaining fields are accessed in TcpConnectTimeout.

It was also found that the checks for all members, except State and CreateTime, could be strengthened by using the advanced specification (the list of internal function calls to target, discussed in 3.2) features of Seance. For the Owner member, it was discovered that it was populated through a call to InetGetClientProcess. This function returns a pointer to the _EPROCESS structure of the process responsible for creating the connection. The disassembly of this flow is shown in Figure 3.3. Note that the rax register is used as the return address of 64-bit function calls. To ensure that the time value is always populated, InetGetClientProcess was included as a function call to find inside TcpCreateAndConnectTcb.

For the members that track the address family and local and remote IP addresses and ports, a calling function specification was created. In particular, during examination of TcpConnectTimeout it was found that these members are passed as parameters to InetFormatSockAddrAtDispathLevel and InetFormatLocalSockAddrAtDispatchLevel.
These functions are responsible for creating `sockaddr_in` structures based on the parameters sent. Figure 3.4 shows commented disassembly of how each member is prepared before both function calls. This allowed for a specification of a function parameter check for `TcpConnectTimeout`.

### Table 3.4 TCP_ENDPOINT members and referencing functions

<table>
<thead>
<tr>
<th>Member</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td><code>TcpCanTcbSend, TcpComputeRtoTcb</code></td>
</tr>
<tr>
<td>AddressFamily</td>
<td><code>TcpConnectTimeout</code></td>
</tr>
<tr>
<td>RemoteIP</td>
<td><code>TcpConnectTimeout</code></td>
</tr>
<tr>
<td>LocalIP</td>
<td><code>TcpConnectTimeout</code></td>
</tr>
<tr>
<td>LocalPort</td>
<td><code>TcpConnectTimeout</code></td>
</tr>
<tr>
<td>RemotePort</td>
<td><code>TcpConnectTimeout</code></td>
</tr>
<tr>
<td>Owner</td>
<td><code>TcpCreateandConnectTcb</code></td>
</tr>
<tr>
<td>CreateTime</td>
<td><code>TcpCreateandConnectTcb</code></td>
</tr>
</tbody>
</table>

```assembly
    call InetGetClientProcess
    ; CODE XREF
    mov    rcx, rax   ; Object
    mov    [rsi+280h], rax
```

**Figure 3.3** Owner being assigned from `InetGetClientProcess`

### Versions Tested

To ensure that this effort covered a substantial variety of modern Windows versions, every major 64-bit Windows 10 version released to date was tested. Table 3.5 provides the OS version and corresponding build numbers covered during testing. Note that this data set covers versions going back to the year 2016 through the latest release as of January, 2021.
Figure 3.4 Several members being accessed from TcpConnectTimeout

Table 3.5 Windows 10 versions and build numbers

<table>
<thead>
<tr>
<th>Version</th>
<th>Build Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1604</td>
<td>10586</td>
</tr>
<tr>
<td>1607</td>
<td>14393</td>
</tr>
<tr>
<td>1704</td>
<td>15063</td>
</tr>
<tr>
<td>1709</td>
<td>16299</td>
</tr>
<tr>
<td>1803</td>
<td>17134</td>
</tr>
<tr>
<td>1810</td>
<td>17763</td>
</tr>
<tr>
<td>1903</td>
<td>18362</td>
</tr>
<tr>
<td>1909</td>
<td>18363</td>
</tr>
<tr>
<td>20H2</td>
<td>19042</td>
</tr>
</tbody>
</table>
Methodology

To begin the testing process, a Seance database for Windows 10 version 17134 was first created. Then, a comparison between the generated database and the information within Volatility 3 was performed. A Volatility 3 plugin was run to ensure connection structures were recovered, along with their complete metadata. Once the data found during binary analysis efforts, Volatility source code review, and Volatility plugin testing was confirmed to be accurate, this was then used as a comparison point for databases generated against all of the tcpip.sys versions.

Creating the databases for tcpip.sys required a few extra steps, as compared to the Objective-C dylibs. First, the PDB files for each version under analysis were required, as opposed to just the binaries. Additionally, the symbol address and names were needed in a parseable format. To accomplish this pdbparse, an open-source tool that can both download and export symbol information, was used, as mentioned in 3.2. Once the symbol information was available, angr could then be provided with the offsets of symbols needed for analysis.

Analysis Results

Analysis revealed several interesting insights. Initially, two issues with version 19042 were observed. First, Seance reported that the symbol chosen for recovering the TCP state, TcpCanTcbSend, did not exist in the PDB. Manual examination of the pdbparse output confirmed this. This required re-analysis of tcpip.sys to find a function present in 19042 that accessed the offset. Subsequent binary analysis found this access in TcpComputeRtoTcb.

The second issue reported was a material CFG change when attempting to recover theCreateTime member. This occurred as all previous versions acquired the time by dereferencing the value at the hardcoded address 0xFFFFFFFF7800000014. This address is not documented by Microsoft, but analyzing cross-references to it gives a strong indication that it is the current system time. Starting with 19042, it was found that the creation time of
connections is instead calculated by calling the \texttt{KeQuerySystemTimePrecise} function. This significant change is what triggered the Seance report.

Looking at the results as a whole, it was found that the offsets of members for tracking the connection state, family, and remote and local IP addresses and ports did not change in any version. It was also noted that these members are near the beginning of the structure. Additionally, it was found that the \texttt{CreateTime} and \texttt{Owner} offsets changed for every version analyzed, but that the offset between the two was always 16 bytes.

\textit{Analysis Conclusions}

The analysis of \texttt{tcpip.sys} showed that, even though the driver is extremely complex, Seance is able to automatically calculate and report both the offsets used to access particular members as well as detect material changes that require manual review. In total, only two changes were required for how particular offsets are calculated compared to the initially generated database - the \texttt{State} and \texttt{Owner} members in version 20H2, as previously described. Furthermore, the necessity for these two changes was automatically detected and reported by Seance.

3.4 Related Work

3.4.1 Data Structure Reconstruction

For the same reasons that motivated this work, there has been significant interest in the recovery of the data structure layout of executable modules analyzed during memory analysis investigations.

\textit{Linux Kernel Analysis}

Previously, [18] described an effort that recovered the offset of several data structure members needed for analysis of Linux processes. This work was groundbreaking, but the techniques
that were used are fragile, since instead of performing full binary analysis, pattern matching was used against the disassembly of functions that access members of interest in a structure. As described in the paper, this approach can only support a limited range of kernel versions.

A very closely-related approach to reconstruct a subset of Linux kernel data structure layouts is taken in [140].

In [117], members of the Rekall development team added support for automatic profile generation from live Linux systems. Prior efforts required the installation of compiler tools on target systems, along with other dependencies. This has obvious negative forensic impacts, so removing that requirement was desirable. Their approach involved pre-compiling abstract syntax trees for mainline kernel versions followed by runtime refinement based on the configuration of the kernel being analyzed. This allowed creation of Rekall profiles capable of analyzing live systems without the need for compiler tools on the target system.

Additionally, [102] describes a system to automatically build Volatility profiles for analysis of Linux memory samples. This project uses a custom Clang plugin to generate kernel source code information and then uses angr to perform symbolic execution against functions in a memory sample. The analysis of these functions reveals the offset of structure members. angr is the standard open source framework for binary analysis, and as discussed in 3.2, is used in Seance as well. This project differs from Seance in key ways though, including 1) it requires the source code of the target module and 2) it is unable to internally detect when its view of a data structure layout is incorrect. The authors also document several instances where Volatility plugins do not produce correct artifacts when using profiles built with their system.

Windows Kernel Analysis

Initial approaches to analysis of Windows memory samples relied on the debugging information of the kernel executable provided by Microsoft [104, 99]. This debugging information is contained within per-executable-version PDB files hosted on Microsoft’s symbol server.
For kernel executables, the PDB will contain the address of all global symbols as well as the layout of each data structure. This satisfies the requirements necessary for structured analysis. As noted in several places, however, there are critical memory artifacts that are not contained within the base kernel module [75, 29, 100]. Most notably are those of associated with the GUI subsystem (\texttt{win32k*.sys}), the network stack (\texttt{tcpip.sys}), and the web server stack (\texttt{HTTP.sys}). For the network and web server stack, the released PDB files have only ever contained the address of global symbols. For the GUI subsystem, PDBs with data structures included were released for a small number of Windows 7 versions, while the rest have only included symbol addresses. These gaps have led to the requirement for significant, manual reverse engineering efforts on the part of memory analysis framework developers.

An approach to solving this issue was documented in [29]. It aimed to add support for a wide range of \texttt{win32k.sys} versions within Rekall. The approach taken was to determine, through manual analysis, which functions referenced members of data structures needed for analysis to succeed. The result was the creation of template files that encode the instructions used to access a particular structure member [126]. To provide a degree of flexibility, the template format wildcards the offsets of control flow redirecting instructions as well as the particular general-purpose register used to store and manipulate values. Unfortunately, this approach is limited in several key ways. First, as noted in the paper, the templates are fragile when it comes to changes in the compiled instruction flow. While more flexible than work before it, the templates require the specific sequence of instructions leading to a member access. This reduces usability across versions of a module, as these frequently change. Second, as discussed in Section 3.3.2, not all accesses to needed structure members occur in the beginning of a function or within a relatively small function. Since the templates require the specific ordering of instructions leading to an access (post-branch unrolling), they are fragile relative to the template’s size.

To alleviate the issues of instruction matching, Seance employs advanced binary analysis that understands the semantics of structure member accesses. This allows it to determine
both if the analyzed function has substantial changes between versions as well as if the
needed member offset(s) have changed.

**Virtual Machine Introspection**

Virtual machine introspection (VMI) is a technique for memory analysis of virtual machine
guests from the host. To perform this analysis, VMI software must meet the same require-
ments as traditional structured memory analysis software. Popular VMI methods to obtain
data structure layouts, or to avoid the issue completely, include graph-based analysis, ma-
chine learning, and code-reuse [56, 44, 69, 111, 45, 38]. Unfortunately, none of these solve
the current problems of memory analysis frameworks as the approaches either require source
code access or they simply borrow and/or re-execute running code inside of a guest to extract
information from an API.

### 3.4.2 Program Analysis in Security

Program analysis, and specifically binary analysis, has long been a focus of security research,
primarily for malware analysis and vulnerability discovery. Projects such as BitBlaze [118]
and angr [55, 116, 121, 115] provide platforms for performing these analysis types.

**Automated Program Analysis**

Polyglot [16], HookFinder [136], and ProspeX [30] attempt to automate specific analysis
tasks. Polyglot focuses on reverse engineering message protocols, and HookFinder detects
and extracts malicious code hooks. These efforts most closely mirror this work, as Seance
also involved building on a general binary analysis platform to automate a specific task.
However, the goals and methods of these projects vastly differ from what is presented here.
**CFG Analysis**

Analysis of control flow graphs (CFGs) is a common technique in malware analysis and defensive security \[97, 15, 23\]. Common applications include comparing CFGs of two pieces of code to determine similarity, and detection of deviation from a known-good CFG to detect return-oriented programming (ROP) exploitation \[26, 116\]. These methods make use of a variety of often sophisticated graph-theoretic techniques to compare CFGs. This work, on the other hand, requires less intense heuristics as it seeks to detect any changes in the graphs of interest, since these indicate a potentially breaking change to forensic tools. This is in contrast to examining purposely malicious software, such as metamorphic malware, which changes many aspects of its CFG between samples.

**Symbolic Execution**

Symbolic execution has been used in numerous research efforts to automatically discover vulnerabilities, generate patches, refine fuzzing efforts, and direct coverage of binary code paths \[17, 34, 130, 135, 95, 64, 121, 24, 131, 37, 61\]. The power of symbolic execution derives from its ability to automatically explore multiple (or all) code paths of a program. Seance leverages symbolic execution to determine when material changes have occurred in a function that will require a manual examination by investigators, but does not attempt to advance the state of the art of symbolic execution.
Chapter Four. Conclusion

Memory forensics is a field which is primarily motivated, driven, and explored by practitioners working on real cases, where large financial stakes, personal safety, or national security may be on the line. At the same time, an investigator’s ability to make sense of a memory image relies on the internal workings of the operating system of the target computer, which can be heavily influenced by academic advances in algorithm and data structure design or privacy and security, and which the investigator may not even have access to the details of. Further complicating the issue is the potential presence of malware on a system. While the idea of "malware" is fairly broadly understood by many, it is in actuality a general and not very well defined term which covers a wide array of behaviors which can be difficult to distinguish from benign behaviors on any given system, leading investigators down deep rabbit holes to try and determine if a computer has been compromised or not.

With the understanding that in memory forensics, the investigator is perhaps the most crucial element, this work addresses several major investigative burdens commonly encountered in real world scenarios, and additionally presents advances in the tools and methods available to memory forensic researchers and investigators. This was done in two parts: firstly, two frameworks, HookTracer and the Seance API, were constructed, based on emulation and symbolic execution, respectively. These frameworks bring powerful tools for program analysis to memory forensics in broadly applicable and extensible ways. Secondly, these frameworks were used to power four tools for addressing specific problems facing investigators. The Volatility plugins hooktracer, hooktracer_messagehooks, and iodine, built on top of HookTracer, help investigators discover malicious behavior on infected systems by exploring the behaviors of API hooks, message hooks, and injected shellcode, respectively. The fourth tool, Seance, automatically discovers changes how given data structures are accessed across different versions of the same binary. Updates to data structures have long plagued forensic tool maintainers, often requiring them to manually reverse engineer bina-
ries, or at least review source code, whenever a new version of relevant software is released. The suite of Volatility plugins were used to successfully identify, and single out, malicious behavior and infected processes on computers infected with several different varieties of malware, including Rovnix, Stuxnet, TDSS, and Zeus, with little to no work on the end of the investigator, aside from running the plugin to gather the data, and running the plugin to analyze it. Seance successfully identified every case where a forensic tool would need to be updated, and, importantly, when it could be left alone, when handle binaries in both the Objective C runtime and in the Windows Kernel, for several different data structures.

4.1 Here Be Dragons

This work opens up several interesting and exciting avenues for future research. As noted in Section 2.3.4, certain malicious behaviors can depend on specific inputs (e.g., looking for something that looks like a Bitcoin public key in a copy and paste buffer before overwriting that buffer [28]), or orders of actions, and as such can be hard to detect with just emulation. In these cases, it seems that a symbolic execution engine could be used in place of HookTracer’s Unicorn emulator to achieve better results, in terms of code paths explored and behavior revealed. Planned future work will attempt just this, by leveraging the Seance API with similar goals to those of hooktracer, hooktracer_messagehooks, and iodine. The hurdles currently identified are getting process memory into a state usable by angr (process memory is several gigabytes in size, too large for angr to handle), and Volatility compatibility. Volatility 2, the version used in this work, works exclusively with Python 2, which is no longer supported, while angr works only with Python 3. This means Seance cannot be integrated as part of a Volatility 2 plugin, as it currently stands. However, Volatility 3 (which uses Python 3) was recently released for public use, and, once it is in a more stable state, should remedy this issue.

Another future project also involves Volatility. As it stands, Seance detects changes
correctly, but only gives an indication that something changed, and still requires investigation if this is the case, no matter how small the discrepancy. Planned future work will attempt to integrate support for automatically detecting changes which require simple or expected updates to offsets, and automatically patching Volatility’s vtypes, data structures inside Volatility which hold the information on how important forensic data structures are laid out, with these updates.

Lastly, with each of the new Volatility plugins instrumenting code, they provide a potential vector for extracting features of the emulated code. By re-configuring HookTracer, or perhaps Seance, to emit data in a way useful to machine learning algorithms, it may be possible to design machine learning-assisted tools for use in the investigative workflow for use in analysing code hooks or injected code.

4.2 Closing Remarks

At the time of writing, all of the work presented here has either been published (hooktracer [20], hooktracer_messagehooks [21]), under peer review (Seance), or is in the process of being written into a paper (iodine). The code used in the projects is currently being reviewed and cleaned up, but will be publicly available on GitHub, with instructions for use, once this is finished. Likely, this will be done under two separate repositories (HookTracer and Seance). Lastly, special thanks to all of the open source projects and developers which made this work possible, especially those involved with angr [55], Unicorn [108], and Volatility [128].
Appendix A: Publishing Agreements

Elsvier Ltd
(Herinafter the "Journal owner")

LICENSE AGREEMENT

PLEASE PROVIDE US WITH THE FOLLOWING INFORMATION, REVIEW OUR POLICIES AND THE LICENSE AGREEMENT, AND INDICATE YOUR ACCEPTANCE OF THE TERMS

Article title: HeckTracer: A System for Automated and Accessible API Hooks Analysis
Corresponding author: Golden G. Richard III
To be published in the journal: Digital Investigations
Your organization: Digital Forensics Research Workshop

YOUR STATUS (PLEASE MARK ALL THAT APPLY)

☐ I am the sole author of the manuscript
   Please indicate which of the below applies to you:
   □ I am a UK, Canadian or Australian Government employee and Crown Copyright is asserted
   □ I am a US Government employee and the Article is public domain and therefore the 'License of publishing rights’ clause does not apply
   □ I am a contractor of the US Government under contract number: .................................................................
   □ None of the above

☐ I am one author signing on behalf of all co-authors of the manuscript
   Please indicate which of the below applies to you:
   □ We are all US Government employees and the Article is public domain and therefore the ’License of publishing rights’ clause does not apply
   □ I am a US Government employee but some of my co-authors are not
   □ I am not a US Government employee but some of my co-authors are
   □ The work was performed by contractors of the US Government under contract number: .................................................................
   □ All or some of the authors are UK, Canadian or Australian Government employees and Crown Copyright is asserted
   □ Some of the authors are employees of the UK, Canadian or Australian Government but Crown Copyright is not asserted
   ☑ None of the above

☐ I am signing as an authorized representative and on behalf of my employer
   Name and job title of licensor if different from corresponding author: .................................................................
   Please indicate which of the below applies to you:
   □ The Article is authored by US Government employees and the Article is public domain and therefore the 'License of publishing rights’ clause does not apply
   □ The work was performed by contractors of the US Government under contract number: .................................................................
   □ My employer is the owner of the copyright in the Article
   □ My employer is the owner of the copyright in the Article (please use the full name of the organization as it should appear in the copyright notice): .................................................................
   □ None of the above

DATA PROTECTION & PRIVACY

I understand that staff of the journal owner and the publisher and its affiliated companies worldwide will be contacting me concerning the publishing of the Article and occasionally for marketing purposes (unless, with respect to such marketing, I tick the box below).

☐ I do not wish to receive news, promotions and special offers about products and services of the journal owner and the publisher and its affiliated companies worldwide
Please tick one or more of the above boxes (as appropriate), review the License Agreement and then sign and date the License Agreement in black ink.

Signed: ___________________________ Name printed: ___________________________

Title and Company (if employer representative): __________________________

Date: 4/25/2019

Please return the completed and signed original of this form by mail or fax, or by e-mailing a scanned copy of the signed original, retaining a copy for your files, to:

Global Rights Department
Elsevier
E-mail: oxfordcopyrights@elsevier.com

---

DIN_458

LICENSE AGREEMENT

License of publishing rights
I hereby grant to the Journal an exclusive publishing and distribution license in the manuscript identified above and any tables, illustrations or other material submitted for publication as part of the manuscript (the “Article”) in print, electronic and all other media (whether now known or later developed), in any form, in all languages, throughout the world, for the full term of copyright, and the right to license others to do the same, effective when the Article is accepted for publication. This license includes the right to enforce the rights granted hereunder against third parties.

Supplemental Materials
With respect to Supplemental Materials that I wish to make accessible through a link in the Article or on a site or through a service of the Journal, the Journal shall be entitled to publish, post, reformat, index, archive, make available and link to such Supplemental Materials on a non-exclusive basis in all forms and media (whether now known or later developed), and permit others to do so. “Supplemental Materials” shall mean additional materials that are not an intrinsic part of the Article, including but not limited to experimental data, e-components, encodings and software, and enhanced graphical, illustrative, video and audio material.

Scholarly communication rights
I understand that I retain the copyright in the Article and that no rights in patents, trademarks or other intellectual property rights are transferred to the Journal. As the author of the Article, I understand that I shall have: (i) the same rights to reuse the Article as those allowed to third party users of the Article under the CC-BY-NC-ND License, as well as (ii) the right to use the Article in a subsequent compilation of my works or to extend the Article to book length form, to include the Article in a thesis or dissertation, or otherwise to use or re-use portions or excerpts in other works, for both commercial and non-commercial purposes. Except for such uses, I understand that the license of publishing rights I have granted to the Journal gives the Journal the exclusive right to make or sub-license commercial use.

User rights
The publisher will apply the Creative Commons Attribution-Noncommercial-NoDerivative Works 4.0 International License (CC-BY-NC-ND) to the Article, where it publishes the Article in the journal on its online platforms on an Open Access basis. For further information, see http://www.elsevier.com/about/open-access/open-access-options.
The CC-BY-NC-ND license allows users to copy and distribute the Article, provided this is not done for commercial purposes and further does not permit distribution of the Article if it is changed or edited in any way, and provided the user gives appropriate credit (with a link to the formal publication through the relevant DOI), provides a link to the license, and that the licensor is not represented as endorsing the use made of the work. The full details of the license are available at https://creativecommons.org/licenses/by-nc-nd/4.0.

Revisions of rights
Articles may sometimes be accepted for publication but later rejected in the publication process, even in some cases after public posting in “Articles in Press” form, in which case all rights will revert to the author (see http://www.elsevier.com/locate/withdrawalpolicy).

Revisions and addenda
I understand that no revisions, additional terms or addenda to this License Agreement can be accepted without the Journal’s express written consent. I understand that this License Agreement supersedes any previous agreements I have entered into with the Journal in relation to the Article from the date hereof.

Copyright Notice
The publisher shall publish and distribute the Article with the appropriate copyright notice.

Author Representations, Ethics and Disclosure Sanctions
I affirm the Author Representations noted below, and confirm that I have reviewed and complied with the relevant Instructions to Authors, Ethics in Publishing policy, Declaration of interest disclosure and information for authors from countries affected by sanctions (Iran, Cuba, Sudan, Burma, Syria, or Crimea). Please note that some journals may require that all co-authors sign and submit Declarations of Interest disclosure forms. I am also aware of the publisher’s policies with respect to retractions and withdrawal (http://www.elsevier.com/locate/withdrawalpolicy).

For further information see the publishing ethics page at http://www.elsevier.com/publishingethics and the journal home page. For further information on sanctions, see http://www.elsevier.com/trade-sanctions

Author Representations

- The Article I have submitted to the journal for review is original, has been written by the stated authors and has not been previously published.
- The Article was not submitted for review to another journal while under review by this journal and will not be submitted to any other journal.
- The Article and the Supplemental Materials do not infringe any copyright, violate any other intellectual property, privacy or other rights of any person or entity, or contain any libellous or other unlawful matter.
- I have obtained written permission from copyright owners for any excerpts from copyrighted works that are included and have credited the sources in the Article or the Supplemental Materials.
- Except as expressly set out in this License Agreement, the Article is not subject to any prior rights or licenses and, if any or any of my co-authors’ institutions has a policy that might restrict my ability to grant the rights required by this License Agreement (taking into account Scholarly communication rights permitted hereunder), a written waiver of that policy has been obtained.
- If I and/or any of my co-authors reside in Iran, Cuba, Sudan, Burma, Syria, or Crimea, the Article has been prepared in a personal, academic or research capacity and not as an official representative or otherwise on behalf of the relevant government.
- If I am using any personal details or images of patients, research subjects or other individuals, I have obtained all consents required by applicable law and complied with the publisher’s policies relating to the use of such images or personal information. See http://www.elsevier.com/patientphotographs for further information. Any software contained in the Supplemental Materials is free from viruses, contaminations or worms.
- If the Article or any of the Supplemental Materials were prepared jointly with other authors, I have informed the co-author(s) of the terms of this License Agreement and that I am signing on their behalf as their agent, and I am authorized to do so.

For information on the publisher’s copyright and access policies, please see http://www.elsevier.com/copyright.

ADDENDUM TO LICENSING AGREEMENT

Notwithstanding anything to the contrary therein, and in addition to those rights set forth in the Licensing Agreement, the Digital Forensics Research Workshop (DFRWS) has the right to host the Article on its website under the CC BY-NC-ND license.

Copyright agreement for "HookTracer: A System for Automated and Accessible API Hooks Analysis", referenced in Chapter 2, sections 2.1 and 2.2
Publishing Agreement

Elsevier Ltd

Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics

Corresponding author
E-mail address
Journal
Article number
Our reference
Pill
DOI

Professor Golden C Richard
goldenrichard@gmail.com
Computers & Security
101872
CSE_101872
SB167-2016(2010)145-0
10.1016/j.cose.2020.101872

Your Status

• I am one author signing on behalf of all co-authors of the manuscript

Data Protection & Privacy

I do not wish to receive news, promotions and special offers about products and services from Elsevier Ltd and its affiliated companies worldwide.

Assignment of Copyright

I hereby assign to Elsevier Ltd the copyright in the manuscript identified above (where Crown Copyright is asserted, authors agree to grant an exclusive publishing and distribution license) and any tables, illustrations or other material submitted for publication as part of the manuscript (the "Article"). This assignment of rights means that I have granted to Elsevier Ltd, the exclusive right to publish and reproduce the Article, or any part of the Article, in print, electronic and all other media (whether now known or later developed), in any form, in all languages, throughout the world, for the full term of copyright, and the right to license others to do the same, effective when the Article is accepted for publication. This includes the right to enforce the rights granted hereunder against third parties.

Supplemental Materials

"Supplemental Materials" shall mean materials published as a supplemental part of the Article, including but not limited to graphical, illustrative, video and audio material.

With respect to any Supplemental Materials that I submit, Elsevier Ltd shall have a perpetual worldwide, non-exclusive right and license to publish, extract, reformat, adapt, build upon, index, redistribute, link to and otherwise use all or any part of the Supplemental Materials in all forms and media (whether now known or later developed), and to permit others to do so.
Research Data

"Research Data" shall mean the result of observations or experimentation that validate research findings and that are published separate to the Article, which can include but are not limited to raw data, processed data, software, algorithms, protocols, and methods.

With respect to any Research Data that I wish to make accessible on a site or through a service of Elsevier Ltd, Elsevier Ltd shall have a perpetual worldwide, non-exclusive right and license to publish, extract, reformat, adapt, build upon, index, redistribute, link to and otherwise use all or any part of the Research Data in all forms and media (whether now known or later developed) and to permit others to do so. Where I have selected a specific end user license under which the Research Data is to be made available on a site or through a service, the publisher shall apply that end user license to the Research Data on that site or service.

Reversion of rights

Articles may sometimes be accepted for publication but later rejected in the publication process, even in some cases after public posting in "Articles in Press" form, in which case all rights will revert to the author (see https://www.elsevier.com/about/our-business/policies/article-withdrawal).

Revisions and Addenda

I understand that no revisions, additional terms or addenda to this Journal Publishing Agreement can be accepted without Elsevier Ltd’s express written consent. I understand that this Journal Publishing Agreement supersedes any previous agreements I have entered into with Elsevier Ltd in relation to the Article from the date hereof.
Author Rights for Scholarly Purposes

I understand that I retain or am hereby granted (without the need to obtain further permission) the Author Rights (see description below), and that no rights in patents, trademarks or other intellectual property rights are transferred to Elsevier Ltd. The Author Rights include the right to use the Preprint, Accepted Manuscript and the Published Journal Article for Personal Use and Internal Institutional Use. They also include the right to use these different versions of the Article for Scholarly Sharing purposes, which include sharing:

- the Preprint on any website or repository at any time;
- the Accepted Manuscript on certain websites and usually after an embargo period;
- the Published Journal Article only privately on certain websites, unless otherwise agreed by Elsevier Ltd.

In the case of the Accepted Manuscript and the Published Journal Article the Author Rights exclude Commercial Use (unless expressly agreed in writing by Elsevier Ltd.), other than use by the author in a subsequent compilation of the author’s works or to extend the Article to book length form or re-use by the author of portions or excerpts in other works (with full acknowledgment of the original publication of the Article).

Author Representations / Ethics and Disclosure / Sanctions

I affirm the Author Representations noted below, and confirm that I have reviewed and complied with the relevant Instructions to Authors, Ethics in Publishing policy, Declarations of Interest disclosure and information for authors from countries affected by sanctions (Iran, Cuba, Sudan, Burma, Syria, or Crimea). Please note that some journals may require that all co-authors sign and submit Declarations of Interest disclosure forms. I am also aware of the publisher's policies with respect to retractions and withdrawal (https://www.elsevier.com/about/our-business/policies/article-withdrawal)

For further information see the publishing ethics page at https://www.elsevier.com/about/our-business/policies/publishing-ethics and the journal home page. For further information on sanctions, see https://www.elsevier.com/about/our-business/policies/trade-sanctions

Author representations

- The Article I have submitted to the journal for review is original, has been written by the stated authors and has not been previously published.
- The Article was not submitted for review to another journal while under review by this journal and will not be submitted to any other journal.
- The Article and the Supplemental Materials do not infringe any copyright, violate any other intellectual property, privacy or other rights of any person or entity, or contain any libellous or other unlawful matter.
- I have obtained written permission from copyright owners for any excerpts from copyrighted works that are included and
Copyright agreement for "Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics", referenced in Chapter 2, sections 2.1 and 2.3
References


[57] Dell Secureworks Counter Threat Unit Threat Intelligence. “Skeleton Key Malware Analysis”. In: (2015).


Vita

Ryan Dominick Maggio was born in New Orleans, Louisiana. He graduated with his B.S. in Mathematics, with a minor in Computer Science, from Louisiana State University in 2016. His experiences with programming, as well as an exposure to the field of computer security, led him to pursue his doctorate in Computer Science at Louisiana State University the following year, with a focus on cybersecurity.