
Masters Theses

Student Theses and Dissertations

Spring 2010

EtherAnnotate: a transparent malware analysis tool for integrating dynamic and static examination

Joshua Michael Eads

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses



Part of the [Computer Sciences Commons](#)

Department:

Recommended Citation

Eads, Joshua Michael, "EtherAnnotate: a transparent malware analysis tool for integrating dynamic and static examination" (2010). *Masters Theses*. 4762.

https://scholarsmine.mst.edu/masters_theses/4762

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

ETHERANNOTATE: A TRANSPARENT MALWARE ANALYSIS TOOL FOR
INTEGRATING DYNAMIC AND STATIC EXAMINATION

by

JOSHUA MICHAEL EADS

A THESIS

Presented to the Faculty of the Graduate School of
MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2010

Approved by

Dr. Ann Miller, Advisor
Dr. Bruce McMillin
Dr. Daniel Tauritz

Copyright 2010
Joshua Michael Eads
All Rights Reserved

ABSTRACT

Software security researchers commonly reverse engineer and analyze current malicious software (malware) to determine what the latest techniques malicious attackers are utilizing and how to protect computer systems from attack. The most common analysis methods involve examining how the program behaves during execution and interpreting its machine-level instructions. However, modern malicious applications use advanced anti-debugger, anti-virtualization, and code packing techniques to obfuscate the malware's true activities and divert security analysts. Malware analysts currently do not have a simple method for tracing malicious code activity at the instruction-level in a highly undetectable environment. There also lacks a simple method for combining actual run-time register and memory values with statically disassembled code. Combining statically disassembled code with the run-time values found in the memory and registers being accessed would create a new level of analysis possible by combining key aspects of static analysis with dynamic analysis.

This thesis presents EtherAnnotate, a new extension to the Xen Ether virtualization framework and the IDA Pro disassembler to aid in the task of malicious software analysis. This new extension consists of two separate components - an enhanced instruction tracer and a graphical annotation and visualization plug-in for IDA Pro. The specialized instruction tracer places a malware binary into a virtualized environment and records the contents of all processor general register values that occur during its execution. The annotation plug-in for IDA Pro interprets the output of the instruction tracer and adds line comments of the register values in addition to visualizing code coverage of all disassembled instructions that were executed during the malware's execution. These two tools can be combined to provide a new level of introspection for advanced malware that was not available with the previous state-of-the-art analysis tools.

ACKNOWLEDGMENT

First, I would like to thank my advisor Dr. Ann Miller for providing the technical guidance, support, and funding necessary to help keep myself focused on the goals of this project and complete this thesis. Her open-mindedness for project ideas allowed me to explore research areas that most likely would not have otherwise been possible.

Next, I must thank Dr. Bruce McMillin and Dr. Daniel Tauritz for their support and feedback throughout my career as a student at Missouri S&T. I have had the delight of taking courses during both of my degrees from Dr. McMillin and have gained a wide array of knowledge and experience in computer security and program analysis from him. I would honestly not be in this position if it were not for meeting and working with Dr. Tauritz throughout my undergraduate and graduate degrees. I completed multiple undergraduate research projects in evolutionary algorithms working with Dr. Tauritz and his Natural Computation Laboratory that guided me to my decision to enroll in graduate school. His interests in computer security and relationship with Sandia National Laboratories has led me on an exciting path to the beginning of a career in computer security research.

Finally and most dearly, I would like to thank my friends and family for all of their support over the years. Particularly I would like to thank my Mom and Dad for putting up with me over all of these years and somehow turning me into a half-decent kid.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
ACKNOWLEDGMENT	iv
LIST OF ILLUSTRATIONS	vii
LIST OF TABLES	ix
 SECTION	
1. INTRODUCTION	1
1.1. BACKGROUND ON MALICIOUS SOFTWARE	1
1.2. BACKGROUND ON MALWARE ANALYSIS	2
1.3. BACKGROUND ON INSTRUCTION TRACING	4
1.4. IMPORTANCE OF PROBLEM	5
1.4.1. Early Response and Defense	5
1.4.2. Attribution of Directed Attacks	6
1.5. CONTRIBUTIONS OF ETHERANNOTATE	7
2. RELATED WORK	8
2.1. XEN ETHER MALWARE ANALYSIS	8
2.1.1. IBM VM/370 Operating System	8
2.1.2. The Xen Hypervisor	9
2.1.3. Ether Malware Analysis Framework	12
2.1.4. VERA Malware Visualization	14
2.2. INSTRUCTION TRACE MALWARE ANALYSIS	15
2.3. CURRENT TOOLS AND ETHERANNOTATE'S ADDITIONS	18
2.3.1. Static Analysis Tools	18
2.3.2. Dynamic Analysis Tools	19
3. METHODOLOGY	21
3.1. ETHERANNOTATE WORKFLOW	21
3.2. XEN ETHER MEMORY MANIPULATION ANALYSIS	22
3.2.1. Hardware and Software for Development	23
3.2.2. Retrieving Instruction Trace from Malware Sample	23
3.2.3. Retrieving Memory Values During Execution	28
3.3. IDA PRO ANNOTATION PLUGIN	29
3.3.1. IDA Python Development Environment	30

3.3.2. Enhancements to IDA Disassemblies	31
3.4. IMPROVEMENTS MADE DURING CASE STUDIES	32
3.4.1. Xen EtherAnnotate Speed Increases	33
3.4.2. IDA Pro EtherAnnotate Porting to C++	35
4. CASE STUDY OF MEGAD BOTNET DROPPER	37
4.1. MALWARE ANALYSIS USING ANNOTATED TRACING	37
4.1.1. Part I - Loading Functions	37
4.1.2. Part II - Decryption and File Dropping	42
4.2. REVIEW OF FINDINGS	44
4.2.1. Benefits of Annotation and Code Coverage	45
4.2.2. The Mysterious Jump Signature	45
4.2.3. Limitations of EtherAnnotate Analysis	47
5. CASE STUDY OF SPYEYE CYBERCRIME TROJAN	48
5.1. FINDING A MALICIOUS BINARY FOR COMPARATIVE ANALYSIS	48
5.2. PREVIOUS ANALYSIS OF SPYEYE TROJAN	51
5.2.1. Trojan Installation	51
5.2.2. Trojan Payload	52
5.3. ETHERANNOTATE ANALYSIS OF SPYEYE TROJAN	52
5.3.1. Initialization and Anti-Tamper Checks	53
5.3.2. Injection into Explorer.exe	54
5.4. COMPARISON OF ANALYSIS TOOLS	55
6. CONCLUSIONS	59
6.1. PRESENTED SOLUTION TO MALWARE ANALYSIS PROBLEM ..	59
6.2. STRENGTHS OF THE CURRENT SYSTEM	59
6.3. ISSUES THAT LIMIT USEFULNESS	61
6.4. FUTURE WORK	63
BIBLIOGRAPHY	65
VITA	69

LIST OF ILLUSTRATIONS

Figure	Page
2.1 Logical Layout of a Type I Hypervisor	10
2.2 Logical Layout of a Type II Hypervisor	10
2.3 Logical Layout of the Xen Hypervisor	11
2.4 Logical Layout of the Xen Ether Framework.....	14
2.5 The Mew Unpacking Loop [1]	16
3.1 The Workflow for Analyzing Malware with EtherAnnotate	22
3.2 Control and Data flow during Ether Single Stepping	25
3.3 Control and Data flow during an EtherAnnotate Call.....	25
3.4 Correlation of the Percent Increase of Filesize and Runtime using Ether- Annotate	30
3.5 Example of Annotated Instructions during a Decryption Routine.....	32
3.6 Area of Xen Ether Source Code Changed for Pausing.....	35
3.7 Area of Xen Ether Source Code Changed for Unpausing.....	35
4.1 The initial decryption loop of MegaD	39
4.2 Final lines of decryption routine and jump to malicious code	39
4.3 Example of the Unpacked Import Table	40
4.4 Jump Sequence after Imports have been Fixed	40
4.5 Using MMX Registers and Instructions to Load Strings	42
4.6 Load all needed DLL exported functions.	43
4.7 Portion of code generating the BAT filename from the current process's filename	43
4.8 The arguments being passed to CreateFileA. Arithmetic operations on register values is currently not supported.....	44
4.9 Example of a call instruction and return instruction in calc.exe	46
5.1 A screenshot of the SpyEye Trojan builder [2]	51

5.2	A screenshot of the SpyEye check for file modifications	53
5.3	A screenshot of the SpyEye injection checks	55
5.4	A screenshot of SpyEye injecting its code into explorer.exe	56
5.5	A screenshot of SpyEye creating a new thread inside explorer.exe	57
5.6	Snapshot of Wireshark Network Trace with Control Experiment.....	58
6.1	Example Instruction Trace from IDA Pro's Debugger	61

LIST OF TABLES

Table	Page
1.1 Overview of Detection Method Effectiveness Against Analysis Tools	4
3.1 Performance Analysis Between Ether Tracing and EtherAnnotate Tracing	28
3.2 Percentage Difference in Time and Speed of Ether and EtherAnnotate....	29

1. INTRODUCTION

There is currently a lack of malicious software analysis applications that allow in-depth, internal information about instruction-level behavior in a single toolset. Individually, dynamic analysis tools provide a fast method for obtaining high-level behavior of an executable while static analysis tools provide a slow, human-intensive method to obtain detailed information of what causes this observed behavior or un-observed behavior. This thesis aims to provide a convergence of these two analysis strategies in EtherAnnotate and demonstrate the ability for new analysis information to be obtained through its use.

This section provides a background on malicious software and malware analysis to help describe the basis for the new research presented. Section 2 provides a context of previous research that this paper extends from. Section 3 describes the new tools that were developed in detail and the design decisions behind them. Section 4 presents a case study where the new analysis tools are used on a recent malicious botnet binary and the benefits and limitations are analyzed. Section 5 presents a second case study which compares EtherAnnotate against current analysis tools during an investigation of the SpyEye Trojan malware. Finally, Section 6 weighs the benefits and limitations of the current research and list potential areas of further development.

1.1. BACKGROUND ON MALICIOUS SOFTWARE

The idea of computer software that is able to secretly replicate itself and execute automatically on newly infected machines originated in the late 1970's with mainframe computers and the early 1980's with personal microcomputers. In 1975, John Walker introduced the program PERVADE on the UNIVAC 1110 mainframe which would secretly copy its caller program (a game called ANIMAL in this case) to all available directories on the system [3]. Seven years later, one of the first modern viruses (although not malicious in intention) was written and distributed by Rich Skrenta for the popular Apple II series personal computer. The software was named Elk Cloner and would hook itself into the reset handler of the Apple II computer, such that when the system was rebooted for the 50th time a poem Rich wrote would appear

on the screen [4]. As personal computers exploded in popularity and knowledge of how to design software that can covertly spread to new machines grew, the number of malicious software attacks in the wild accelerated and became increasingly difficult to defend against.

Today, malware can spread globally to thousands of computers in only a matter of days and performs much more sophisticated and damaging attacks to systems [5]. A new trend in malicious software has been the increased prevalence of botnets - a covert overlay network of personal computers which have been infected by malware and receive orders from a centralized or peer-to-peer “command and control” communication channel. Single botnet variants have been able to infect over a million computers and typically either steal financial information (76% of phishing attacks are for financial gain according to Symantec’s 2009 threat report [6]) from these machines or use them to perform coordinated DDoS attacks against large corporations or government entities [7]. Currently, the most prevalent strategy to mitigate botnet attack potential is to reverse engineer the malicious binaries that are dropped and determine the identities of the command and control servers. Once the identities of command and control servers are known, either the ISP can be contacted to shut the server down or security companies can try to have DNS servers not recognize the malicious IP addresses.

The number of new malicious attacks developed and launched against machines increases every year, becoming more advanced and requiring more skilled analysis to detect and defend against. Additionally, the source of malicious attacks has migrated from devious hackers launching small and simple attacks to criminal organizations and even nation states organizing sophisticated, targeted attacks against their adversaries. Defending against malicious computer attacks has become a tremendous and critical industry and research market and the need for comprehensive analysis and rapid response grows every year.

1.2. BACKGROUND ON MALWARE ANALYSIS

In order to protect their anonymity from security analysts and increase the difficulty of defense, malware authors generally obfuscate their code and their communication channels. A very common method for hiding data from security analysts

is for the malicious code to detect if common analysis tools are being used and change its behavior if they are detected. One such method is to detect changes in system behavior that occur when a software debugger is attached to the running process (anti-debugging). In Windows, attaching a debugger (e.g., OllyDbg or WinDBG) sets a variety of system flags and causes small details in system behavior to change compared to a system without a debugger attached to a process [8]. Malware will commonly detect these changes and either exit without performing any malicious activity or perform suspicious activity to draw the malware analyst away from the real malicious activity. There are anti-anti-debugging tools that can be used with debuggers that try to hide the debugger from detection although these are complicated and only prevent detection from known methods.

Another method malware authors use to detect that their software is being watched is to detect that the code is executing inside a virtual machine (VM). Analysts commonly run malware inside of a virtual machine in order to sandbox the code from infecting the host machine and to enable the ability to “rewind” the state of the system. Malware can detect that it is running inside of a virtual machine by checking for virtualized hardware, differences in certain instruction return values, and backdoor I/O ports used by the hypervisor to communicate to the VM [9]. As virtualization is increasingly utilized today to consolidate physical server hardware into virtual hardware, virtualization detection is not as prevalent debugger detection. This is because *not* performing malicious activity when inside of a VM would eliminate a large percentage of potential victims.

Table 1.2 provides an overview of common analysis detection methods that are able to detect three common analysis tools and how the EtherAnnotate implementation compares. Debugger and virtualization detection are previously described; dynamic analysis detection refers to the detection of any kinds of runtimes analysis tools being used to monitor the malware. Common detection methods for dynamic analysis are searching through the process list for known analysis tool filenames and checking the consistency of API function calls to check for tools that may hook these functions and cause changes in their instructions. OllyDbg is a common instruction-level debugger for applications, Process Monitor is a common dynamic analysis tool

Table 1.1. Overview of Detection Method Effectiveness Against Analysis Tools

	OllyDbg	ProcMon	CWSandbox	EtherAnnotate
Debugger	Yes, with plugins	No	No	No
Virtualization	No	No	Yes	Yes
Dyn. Analysis	No	Yes	Yes, hook detect.	No

for monitoring application behavior, and CWSandbox is a virtualized sandbox environment that hooks API calls to record behavior. EtherAnnotate is imperceptible to any operating system level detection attacks since no modifications are made to the guest OS. Alternatively, the only detection attacks known to the author against EtherAnnotate would have to rely on virtualization timing and device detection for the Xen framework. There have been multiple papers on virtualization detection that mostly focus on VMware [9] and other Type II hypervisors [10], additional research is needed to determine a concise list of methods to detect the Xen 3.x framework.

Lastly, malware authors can obfuscate their code to static analysis by encrypting or scrambling the binary operation codes (opcodes) in the executable. The binary file then has a separate routine that decrypts or unscrambles the obfuscated opcodes so that the original source code can be executed. Analysts can subvert this technique by watching the application with a debugger and dumping the process’s memory after it has been decrypted. Most advanced malware utilize very advanced encryption and scrambling techniques that may involve multiple processes or other obfuscation techniques that deceive disassemblers.

1.3. BACKGROUND ON INSTRUCTION TRACING

The idea of recording the set of all instructions executed during a program or computer’s execution is a technique that started with early computer debugging support in the early 1970’s. Barnes et al. published an early paper that implemented an instruction tracer in the HP 2100A minicomputer’s firmware in order to aid in software debugging [11]. In addition to tracing all instructions executed on a computer, the HP 2100A could selectively trace only instructions occurring in a specific memory area. Since the tracer was implemented in the computer’s firmware, it was able to trace operating system execution and any program’s execution without any

modification to the underlying programs. Instruction tracing has not changed much since the early implementations, although today most instruction trace applications are userland applications (i.e., separated from the OS kernel and unprivileged) that run underneath the operating system. Most common integrated development environments (IDEs) and debuggers contain some sort of instruction or event tracing today.

In a sense, the instruction trace capabilities presented in this thesis are closer to the early HP 2100A tracer than modern tracing applications. The EtherAnnotate instruction tracer runs in an elevated privilege environment much like the firmware on the minicomputer - it runs on the Xen privileged Dom0 environment and communicates solely with the Xen hypervisor. This similarity allows for EtherAnnotate to monitor and trace any operation of a guest operating system, including the operating system's execution itself. Additionally, as each instruction is recorded, the values of currently accessed variables are also recorded in the EtherAnnotate instruction trace log.

1.4. IMPORTANCE OF PROBLEM

New methods that provide a deeper understanding of malicious software as it is discovered are necessary to provide adequate defense against future computer network attacks. A solution needs to be able to quickly analyze new malicious samples and help security analysts deduce how the malware works fast enough to develop a response strategy. Once an initial defense strategy has been formulated, any increased accuracy in attribution of the original authors or groups responsible for the malware will help security labs and government agencies find and eliminate the cause of the threats. Finally, any tools that provide a more detailed understanding of the malware's operations will allow analysts to better predict future attacks and behavior which can lead to proactive defense.

1.4.1. Early Response and Defense. Previous large-scale attacks by worms and botnets have clearly shown the necessity for an early response and deterrent in order to mitigate the possible damage of coordinated attacks [12]. For these globally spreading worms, the previous research focuses mostly on detecting the presence of the large network of victim computers and issuing security notices

about potential attacks and methods for defense. For smaller-scale, targeted attacks with binary payloads it may be more pertinent to instead focus on the intentions of the attack that can be deduced through internal analysis of the malicious payload. Typical dynamic code analysis techniques as described in Section 2.3.2 provide a quick, high level overview of the behavior of a binary during runtime. A lower, more detailed knowledge of a complex malicious binary typically requires a larger amount of time to be invested in static code analysis as described in Section 2.3.1, usually by reverse engineering the assembly language instructions [13]. The EtherAnnotate analysis toolset presented in this thesis provides a convenient method for adding runtime dynamic analysis information to a binary’s disassembled instructions with a minimal number of changes to the computer system.

1.4.2. Attribution of Directed Attacks. The attribution of cyber attacks on computer systems is a growing concern for both corporate security [14] and national security [15][16]. The Department of Defense commissioned report on techniques for attributing cyber attacks defines the term attribution as “determining the identity or location of an attacker or an attacker’s intermediary” [15] and this is one of the core goals of malware analysis. Network attack attribution is particularly difficult because of the inherently redundant and anonymous nature of the Internet and computer systems. Attackers may plant code that waits an extended period of time before causing damage or may work through intermediary machines unrelated to the originating computer system. Simply tracing the IP address of a recorded cyber attack may not provide a target to the attacker’s personal computer (e.g., use of cyber cafes or rerouting the attack through victim machines) and in cases of anonymizing overlay networks it may not even be possible to trace the originating machine [17].

Binary file attribution provides a similar level of difficulty in determining the original author or even a general location of the originating attack. Since malicious binary files are most often spread autonomously, the computer that sent the file to the victim can not reliably be used as the true attacker’s origin. Common techniques for binary attribution rely instead on correlating similar binary files based on common patterns that can be identified between many samples in a large binary library [18]. This attribution, typically using n -gram analysis or Bayesian analysis, builds on the hypothesis that malware written by the same author or authors will contain traits of

those who wrote the original code - even in compiled form. Any uniquely identifying information about a binary file can help with attribution; EtherAnnotate provides a novel method for obtaining internal identifying strings not easily discovered using current analysis methods.

1.5. CONTRIBUTIONS OF ETHERANNOTATE

The EtherAnnotate toolset presented in this thesis addresses the issues of providing a quick, in-depth analysis of malicious software by contributing a novel unification of dynamic analysis and static analysis that is largely transparent to detection. The EtherAnnotate toolset consists of two independent tools - an extension to the Xen Ether malicious software analysis framework described in Section 3.2 and a plug-in to the IDA Pro graphic disassembler described in Section 3.3. Previous static analysis tools lack a transparent method for viewing the runtime data of a program's execution in a visually coherent manner. The Ether additions allow for an instruction tracer running outside of a virtual machine to monitor and transparently record the variable values of a program while it executes. The IDA Pro plug-in takes the output from the first tool and uses the variable values to annotate a graphical disassembly of the malicious software's code. Code coverage visualization is accomplished inside of IDA Pro by highlighting all instruction addresses listed in the EtherAnnotate trace file. By combining the statically disassembled instructions of a malicious program with annotations of its actual runtime values, this new visualization provides an increased awareness of the program's operations while a researcher performs static code analysis. The author believes that this new toolset will contribute additional knowledge to malware analysts and provide a starting point for additional research in malware behavior analysis. Additionally, the later case studies in Section 4 and Section 5 provide not only a comparison of analysis results from EtherAnnotate and previous techniques, but also internal functionality insight of two popular malicious applications.

2. RELATED WORK

The work completed in this thesis integrates new functionality into previous research done on the Xen virtualization framework and the Ether analysis framework. The Xen Ether framework was originally designed as a malware analysis tool that allows instruction tracing, system call tracing, and automatic unpacking of malicious binaries. This research extends the Ether framework to also record all processor register memory access and supplement IDA Pro disassemblies with that information. A review of recent research into botnet protocol reverse engineering and static/dynamic analysis will help illustrate the design choices of this thesis. Sections 2.1 and 2.2 present previous areas of research that this thesis builds off of and extends. Section 2.3 presents the current tools available both in research and production for analyzing complex malware that EtherAnnotate seeks to improve upon.

2.1. XEN ETHER MALWARE ANALYSIS

2.1.1. IBM VM/370 Operating System. The idea and usefulness of running multiple, different operating systems on a single computer system was first realized in the 1970's by researchers at IBM [19]. The IBM VM/370 system split the computer's operating system into two components, the control program (CP) and the Conversational Monitor System (CMS). The CP ran underneath all virtualized operating systems and provided resource management and control to all of the virtual machines. The CMS provided a virtual terminal for each virtual machine that the user could interface with and appeared as though they were interfacing with a bare-metal operating system. A key design criteria for the VM/370 software was that all virtual operating systems running on top of the CP would execute the same instructions as if they were running on the machine by themselves. Additionally, all operations in the virtual machines were guaranteed to provide the same output if they were running in a virtual environment as if they were running on a physical processor.

At the time, the VM/370 operating system was noted for its novel ability to allow multiple users concurrent access to a single computer and the added security provided by address space isolation between the virtual machines. Unfortunately,

the idea of virtualization did not move outside of IBM's mainframes and into the personal computing sector until the last decade. IBM's VM/370 operating system ran on top of its System/370 mainframe which was designed to allow virtual machines to execute on the processor without interference [20]. One vital design aspect that came from the System/360 hardware was to divide all instructions into two mutually exclusive sets: privileged and unprivileged instructions. Privileged instructions affect the operations of the entire machine while unprivileged instructions only cause local changes and are commonly called by user applications. The System/370 hardware required all privileged instructions to be handled by a specific program, and the VM/370's Control Program transparently handled these instructions for its virtual machines. Additionally, the System/370 hardware was designed to support virtual memory from its introduction and allowed for the control program to imperceptibly share physical memory between the virtual machines. The x86 hardware, which has become the standard processor architecture in personal computing, was not designed to support virtual machines and many workarounds have been developed in recent years to enable virtualization on PCs.

2.1.2. The Xen Hypervisor. In 2003, a research group based at the University of Cambridge published Xen, an open source x86 virtual machine monitor (VMM) which (eventually) allowed unmodified operating systems to run in isolated virtual machines [21]¹. Xen now supports x86-64, Itanium, and the PowerPC architectures and is one of the most popular open source virtualization solutions today. Although it was designed around 30 years after the original VM/370 software was written, Xen shares many core similarities to IBM's original virtualization solution. The Xen framework consists of a Type I hypervisor which runs in a bare-metal configuration below all operating systems and virtual machines. Goldman's original paper on computer virtualization describes two different types of hypervisors that still hold true with present solutions [22]. A Type I hypervisor runs on the bare hardware and all virtual operating systems on the machine run virtualized on top of the hypervisor as in Figure 2.1. A Type II hypervisor allows a standard operating system to run on the bare hardware first, then the hypervisor is loaded on top of the operating system and runs virtual machines from this higher level as in Figure 2.2.

¹This thesis will use the terms *virtual machine monitor* (VMM) and *hypervisor* interchangeably

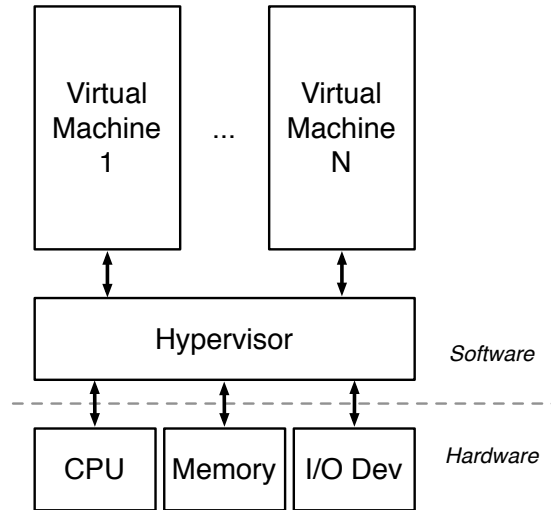


Figure 2.1. Logical Layout of a Type I Hypervisor

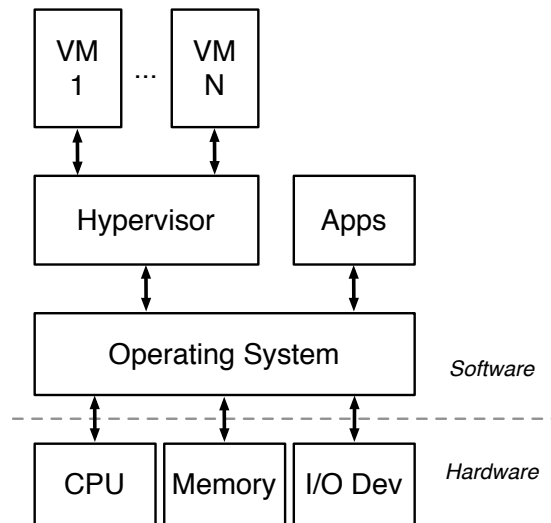


Figure 2.2. Logical Layout of a Type II Hypervisor

The Xen framework modifies the standard Type I hypervisor design such that there is a single privileged virtual machine named Domain 0 (Dom0) and all other virtual machines, called unprivileged guest domains (DomU), act like standard virtual

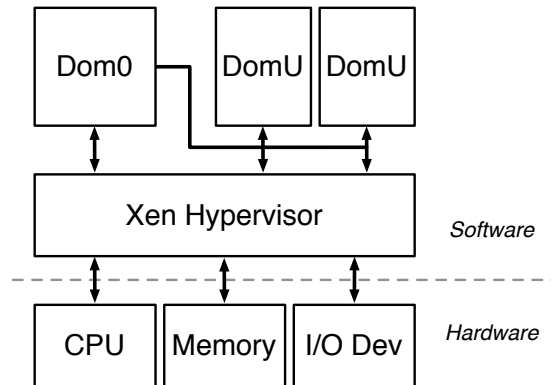


Figure 2.3. Logical Layout of the Xen Hypervisor

machines in other frameworks. Dom0 is the first domain to boot after the hypervisor is loaded by the computer’s BIOS, and provides an administrative interface to the Xen hypervisor [23]. Typically, Dom0 is a modified Linux kernel that appears to be a normal Linux OS with the addition of a list of administrative commands that can instruct the hypervisor to create new VMs, modify existing VMs, shutdown existing VMs, or perform other hypervisor commands. All of the regular VMs will be spawned from Dom0 and run on top of the Xen hypervisor and communicate through the emulated devices created in Dom0 as shown in Figure 2.3.

In 2006, AMD released processors with its new AMD-V hardware virtualization extensions and Intel released a similar new feature in its Core 2 processors named VT-x. These extensions to the x86 specification implement additional functionality that allows the x86 processor to fulfill the classical virtualization requirements. In 1974, Popek and Goldberg published the seminal paper on the formal requirements of a processor instruction set to be virtualizable; in it they state that an instruction set must allow for *equivalence*, *resource control*, and *efficiency* with regard to a virtual machine [24]. While previous x86 virtualization techniques had to resort to clever software design to emulate equivalence and handle resource control, the new virtualization extensions provide additional controls that are based on Popek and Goldberg’s description of how to design a virtual machine monitor.

The VT-x extensions create a new execution mode called VMX mode which in turn has two separate privilege levels, VMX root mode and VMX non-root mode. VMX root mode was designed as the mode that a hypervisor would execute in and thus is very similar to execution outside of VMX mode other than the addition of a new set of VMX instructions and access to new VMX storage variables. In VMX non-root mode, the processor execution is restricted so that certain privileged instructions (as defined in Popek and Goldberg [24]) are trapped and cause a new event called a VMEXIT. By causing VMX non-root privileged instructions to trap to a predetermined location in hardware, the VT-x extensions now allow the x86 hardware to provide the *dispatcher* and *interpreter* aspects of a VMM natively. The VMM in root mode can configure the processor to trap on more instructions and actions than set by default; additional traps can be set for if a guest tries to access certain I/O ports or control registers in the processor [25]. Because VMX mode splits execution into root and non-root modes, a virtualized operating system can be run unmodified and with its kernel operating in current privilege level (CPL) 0. All modern operating systems run their kernel in CPL 0 and all applications in CPL 3 which allows the hardware to restrict user-mode applications from executing privileged instructions [26]. Before virtualization extensions were introduced to x86, VMMs would have to make do by typically running the VM kernel in CPL 1 and the VM applications in CPL 2; however, VMX root and non-root modes now have a separate CPL for each.

Xen 3.0 added support for Intel's VT-x technology which is required and used extensively by the Xen Ether framework. These extensions allow fine-grained control of the hypervisor with hardware guarantees of transparency for the equivalence and resource control criteria.

2.1.3. Ether Malware Analysis Framework. In 2008, Dinaburg et al. published a paper describing their new framework called Ether which builds a malware analysis tool into the Xen hypervisor [27]. Their framework runs solely on the Xen hypervisor and Dom0 without requiring any modifications to the virtualized guest operating system in DomU. The separation of the analysis framework and the guest operating system is furthered by extensively utilizing the Intel VT virtualization extensions. Ether requires the use of Xen 3.0 and hardware-assisted virtualization (HVM) guests which allows for the highest level of transparent malware analysis

without the guest being able to detect analysis. Similar analysis using guest-based virtualization (e.g., VMware [28] and VirtualBox [29]) or kernel-level analysis (e.g., VAMPiRE and CWSandbox) are not able to meet all of the transparency requirements presented. In the paper, the authors present five requirements of a transparent malware analyzer that the Ether framework is able to fulfil:

- **Higher Privilege:** The malware analyzer must run at a higher hardware privilege level than the guest operating system it is analyzing. Ether runs at the highest privilege level, CPL -1 using Intel VT, while the guest operating system’s kernel and userspace run at CPL 0 and 3 respectively.
- **No Non-Privileged Side Effects:** The malware analyzer must not induce any side-effects in the guest operating system that a malicious program running in the guest can detect. Ether has protected access to certain processor registers and has isolated memory access to the guest through shadow page tables.
- **Identical Basic Instruction Execution Semantics:** All instructions executed during analysis must have the same behavior as if they were running directly on the processor (similar to Popek and Goldberg’s equivalence requirement). Ether uses the Intel VT virtualization extensions which rely on the same hardware execution semantics as non-virtualized execution.
- **Transparent Exception Handling:** All exceptions must behave and return during malware analysis as they would during normal execution. Ether only traps certain exceptions, does not cause any new guest-detectable exceptions, and does not modify the returned values of guest-generated exceptions.
- **Identical Measurement of Time:** A common attack to detect if a guest is inside of a virtual machine is to measure the time required to execute a set of instructions. If the analysis framework does not have a *privileged* clock that it can modify to negate extra time taken for analysis and VM exits, then the guest can detect the presence of virtualization. Intel’s VT extension provides a separate execution cycle count in the hypervisor which can be used to offset the logical clock return values.

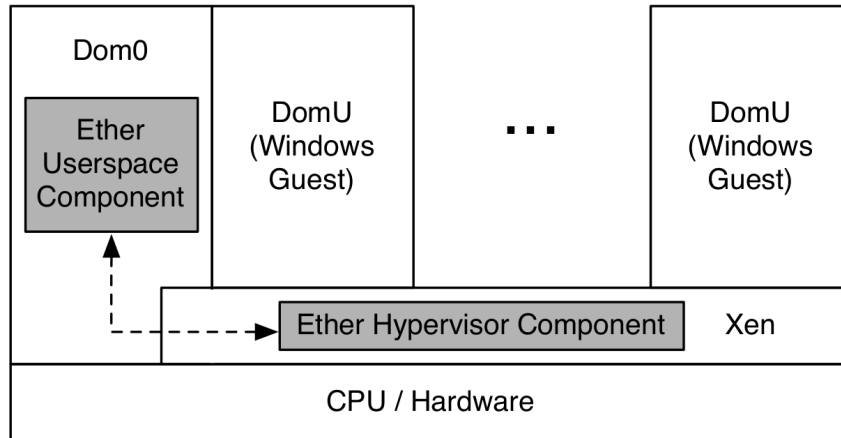


Figure 2.4. Logical Layout of the Xen Ether Framework

Once the Ether framework has been shown to provide transparent analysis into a guest operating system, the authors present the implementation and examples of analysis tools built on Ether. The main component of Ether, as seen in Figure 2.4, runs in the Dom0 privileged Xen domain and provides user interaction to the analysis tools. A small number of changes were made to the Xen hypervisor code to provide abilities to trap different operations in the guest operating system. The main analysis tools are split into instruction/memory write tracing and automated unpacking.

Ether implements instruction tracing of the guest operating system by setting the trap flag after each instruction which causes a debug exception and a VM-exit event that the Xen hypervisor handles. The Ether analysis framework receives control of the system after debug exceptions induced by the trap flag being set which allows EtherAnnotate to perform its analysis and instruction trace logging in between each guest instruction. The process Ether uses to set the guest's trap flag and give EtherAnnotate control of the system is thoroughly documented in Section 3.2.2.

2.1.4. VERA Malware Visualization. In 2009, Quist et al. published one of the first research tools which extended the Xen Ether analysis framework - particularly to increase the speed at which malware analysts can deobfuscate a binary [1]. The Visualization of Executables for Reversing and Analysis (VERA) architecture the authors present takes a standard instruction trace file from Ether

combined with the original malicious binary to generate a visualization of code flow during execution. Figure 2.5 from the original VERA paper depicts an example of running VERA on the unpacking loop of a binary packed with the Mew packer. The visualization is an abstraction of the instruction trace that Ether recorded of the Mew-packed binary where each node represents a basic block (list consecutive instructions until branch) and each edge represents a branch to a different basic block which was taken. The green nodes represent basic blocks of instructions which are non-existent in the original binary, such as areas that have been allocated on the heap or are self-modifying. The red nodes represent sections with high entropy, such as when the binary is executing in newly unpacked areas of its memory (since the original memory was packed and has high entropy, cf. encrypted data entropy). In the Mew packer example, code flow originally loops in an initial unpacking loop which appears to unpack the second, larger loop since the second loop executes in an area of high entropy. The second unpacking loop is much more complicated than the first from the visual inspection of many more branches and longer loops. Eventually, it appears that the second unpacking loop allocates memory to place the original code into and progresses into the original entry point (OEP).

Quist mentions that although there are similar programs available which provide graph-based analysis, Ether and VERA are unique in that they do not require the use of debuggers or guest state modifications. This allows VERA to provide analysis transparently to the guest operating system and any malware that may try to detect debuggers or analysis tools [8].

2.2. INSTRUCTION TRACE MALWARE ANALYSIS

A research paper by Sharif et al. was published in 2009 on a similar project in that it uses protected instruction tracing as the foundation for malware analysis [30]. In their paper, the authors present a system called Rotalum  which executes malware in a protected environment, records an instruction trace, and uses data-flow and taint analysis to automatically reverse engineer emulator syntax and semantics. The paper focuses on malware that has been protected using the increasingly common obfuscation technique of emulation (common tools are Themida [31] and VMProtect [32]) . An emulation protection scheme for binary obfuscation translates the original binary's

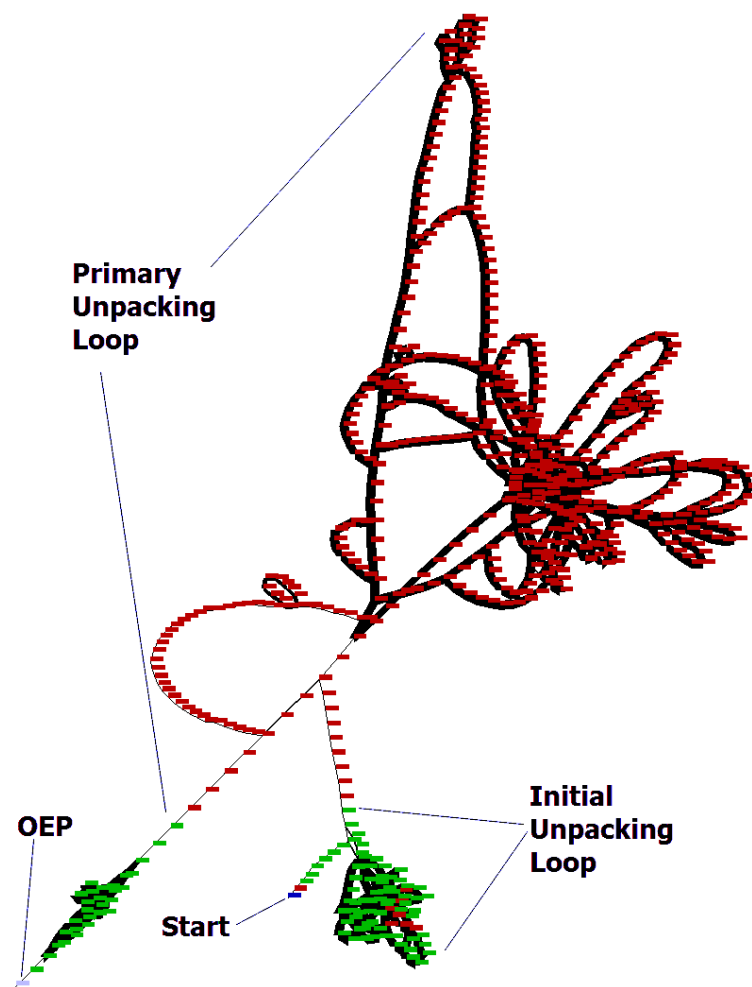


Figure 2.5. The Mew Unpacking Loop [1]

x86 instructions into bytecode instructions that can be interpreted by a program virtual machine at runtime. This technique is similar to how Java programs are compiled from their Java source code to an intermediate bytecode language that is executed by the Java Virtual Machine (JVM) which translates the bytecode instructions into the current architecture's opcodes. Once a program has been protected using one of these emulator protection tools, the malware analyst only has a binary file with the protector's emulator code and the malware's bytecode which has unknown syntax and semantics for most of the common protectors.

The Rotalumé tool that Sharif et al. developed consists of three core strategies: identifying the virtual program counter (VPC), identifying emulation behavior, and extracting the syntax and semantics of the emulator. The authors developed a new technique they call *abstract variable binding* which attempts to identify program variables used during the execution of the program from an instruction trace. Their goal is to find the VPC variable that emulators use as an index into the protected executable’s bytecode to find the next instruction to execute, such as:

```
instruction = bytecode[VPC]
           or
           instruction = *VPC
```

The authors assume that most emulator protection methods will use this instruction fetching technique and develop a series of algorithms and rules for identifying possible VPCs and using clustering to narrow the field down. Once the candidate VPCs are narrowed down, the Rotalumé system uses dynamic taint analysis to find execution read behavior that is common to decode-dispatch emulators. For each potential VPC found in the first section, the second section does 2nd-pass analysis on all other instructions to see how the VPC memory is read and used by other instructions. Once a loop has been detected that executes at least twice and appears to be decoding bytecode using the candidate VPC and dispatching execution then that cluster is identified as the correct VPC cluster. Finally, the system monitors how the execution proceeds after a dispatch has been made and looks for bytecode instructions that change the VPC to try to identify branching instructions. Identifying control flow transfers allows the authors to build control-flow-graphs that assist in analysis of the behavior of the emulator’s bytecode language.

EtherAnnotate began development around the time that the paper on Rotalumé was published and uses some of the same ideas about process introspection from an overlaid system, but focuses on the angle of internal algorithm analysis. Where Rotalumé focuses more on the automation of determining emulator protection semantics, EtherAnnotate was developed as a tool that can provide run-time variable information in a static analysis environment with a guarantee of transparency. Further work

combining the variable identification techniques of Rotalumé and EtherAnnotate in a second-pass analysis could provide more easily understood information about what values were stored in program variables at run-time.

2.3. CURRENT TOOLS AND ETHERANNOTATE'S ADDITIONS

Current tools for malware analysis are divided into static analysis tools and dynamic analysis tools. Static analysis describes a process of observing a program without actually executing any of the code - this provides a safer method of analysis with malware and usually provides more detailed information but lacks knowledge of actual execution behavior. Dynamic analysis is the process of executing a program and monitoring any aspect of its behavior - this method typically provides a faster understanding of the broad characteristics of a program but can skip over many of the program's inner workings. EtherAnnotate aims to provide a collection of benefits from both static and dynamic analysis in a single package while eliminating many of the detection issues caused by using intrusive dynamic analysis tools such as debuggers.

2.3.1. Static Analysis Tools. Static code analysis tools provide information about a binary file without executing the file, therefore providing a static image of the actual code and data contained within the file. The most common static analysis tool is a disassembler which takes the processor opcode instructions that are stored as raw data within the executable binary file and converts these opcode datum into the assembly language mnemonics that are human-readable. Disassemblers provide an accurate representation of what instructions the processor will execute when the binary file is loaded by the operating system. However, since the x86 instruction set has a variable-length set of instructions, there can be multiple disassemblies possible and some more advanced disassemblers are able to detect anti-disassembly procedures and produce the correct disassembly. In addition to disassemblers, there is another class of static analysis tools called decompilers which try to convert an executable's assembly code into a reasonable representation in a high-level language such as C. These tools are much less accurate than disassemblers since there are a vast magnitude of ways a compiler can take a high-level source code file and compile it down to assembly code. Additionally, compilers typically optimize their compiled assembly language code for either increased speed or decreased size - these optimizations can

make it more difficult to recreate the original source code as a side effect. The following list is a brief summary of static analysis tools that are commonly used in malware analysis:

- **Ollydbg:** One of the more common debugging tools, OllyDbg can also be used for its disassembling abilities although it does require the application under analysis to be executed up to a point. OllyDbg also has wide 3rd party support and there are many plug-ins for hiding from malware, dumping a program's memory contents to a file, and other uses.
- **IDA Pro:** A disassembler that recently added debugging functionality, IDA Pro provides a graphical representation of the disassembled code that separates it from other disassemblers. There is also a 3rd party plug-in API for C++ and Python that is used for EtherAnnotate's visualization tool.
- **Hex-Rays:** A decompiler plug-in for IDA Pro, Hex-Rays is one of the more advanced decompilers and will convert disassembled instructions into C-like pseudocode.
- **PEiD:** Using a list of signatures from various packing tools and compilers, PEiD provides identification information about binary executables. DLL and EXE files can be loaded by PEiD and if there is a known signature for the program that compiled the binary or the program that packed the binary, PEiD will report that information.

2.3.2. Dynamic Analysis Tools. In contrast to static code analysis tools, dynamic analysis tools provide a method for monitoring and analyzing how a program behaves during its execution. Dynamic analysis typically provides a quicker method for obtaining information about an executable but it also is typically not as detailed as what can be obtained through static analysis. One type of tool that straddles both dynamic and static analysis is a debugger. Debuggers for malware analysis work with the disassembled instructions of a binary file and allow the analyst to step through each line of execution, view the active memory of a process, set breakpoints for the execution to pause at, and other useful run-time modifications. Most other dynamic analysis tools provide some sort of monitoring about the program, be it file activity,

network activity, system call logging, or other attributes that may define a program's behavior. The list below is a brief summary of common dynamic analysis tools that are commonly used in malware analysis:

- **OllyDbg:** The main purpose of OllyDbg is as a assembly-level debugger and it is one of the most popular applications in this regard. The program provides access to all internal data structures of a process and allows for a variety of breakpoints to be set based on different conditions.
- **ProcMon:** There are many different events a process can cause during its execution and Process Monitor (ProcMon) allows the malware analyst to track most possible events. Some examples of process events are file system manipulations, registry manipulations, and certain Windows API system calls.
- **oSpy:** This program will hook certain Windows API system calls related to sending network traffic and record both the contents of the function arguments as well as the call stack to each system call. A conversation log can be generated with the contents of each network message sent or received.
- **RegShot:** The Windows registry provides configuration data for most system operations and applications on the operating system. Regshot takes a snapshot before and after a program executes and provides a means for showing differences in the registry that occurred.

3. METHODOLOGY

This section presents the design decisions and implementation details of the EtherAnnotate plug-in for Xen Ether (Section 3.2) and IDA Pro (Section 3.3). Additionally, an explanation of the intended workflow of a malware analyst using the EtherAnnotate toolset is presented in Section 3.1. Section 3.2.2 provides a secondary contribution of insight into how the virtual guests, Xen, and Xen Ether communicate and the code flow between them as learned during the development of EtherAnnotate. In addition to the topics covered in this section, further methodology design changes were made during the second case study and these changes are presented in Section 3.4.

3.1. ETHERANNOTATE WORKFLOW

The EtherAnnotate toolset was designed for use by a malware analyst who wants to gain insight into the inner-workings of advanced malicious software as it would behave on a physical system. The proposed solution as shown in Figure 3.1 utilizes two common software packages - Xen for virtualization and IDA Pro for disassembly visualization - and provides a simple workflow that only requires three main tasks to complete. The first task of the workflow is to take a binary file (that may or may not be known to be malicious) and execute the file while under the EtherAnnotate environment. As the figure and Section 3.2 describe, the EtherAnnotate environment is an analysis tool which is built upon the previously developed Ether and Xen toolsets. As the binary is executed, a log file is produced which contains all instructions that were executed, a list of all register values referenced in the instructions, and a list of all possible string array values referenced. This file will be the EtherAnnotate specialized instruction trace used for the visualization process.

The second main task is simply for the malware analyst to transfer the EtherAnnotate instruction trace file to a workstation that had IDA Pro and the EtherAnnotate plug-in installed. The final workflow task is for the analyst to load an unpacked version of the original binary file into IDA Pro. Binary packing and unpacking is detailed in Section 1.2 and there are well known methods to attempt unpacking such as using

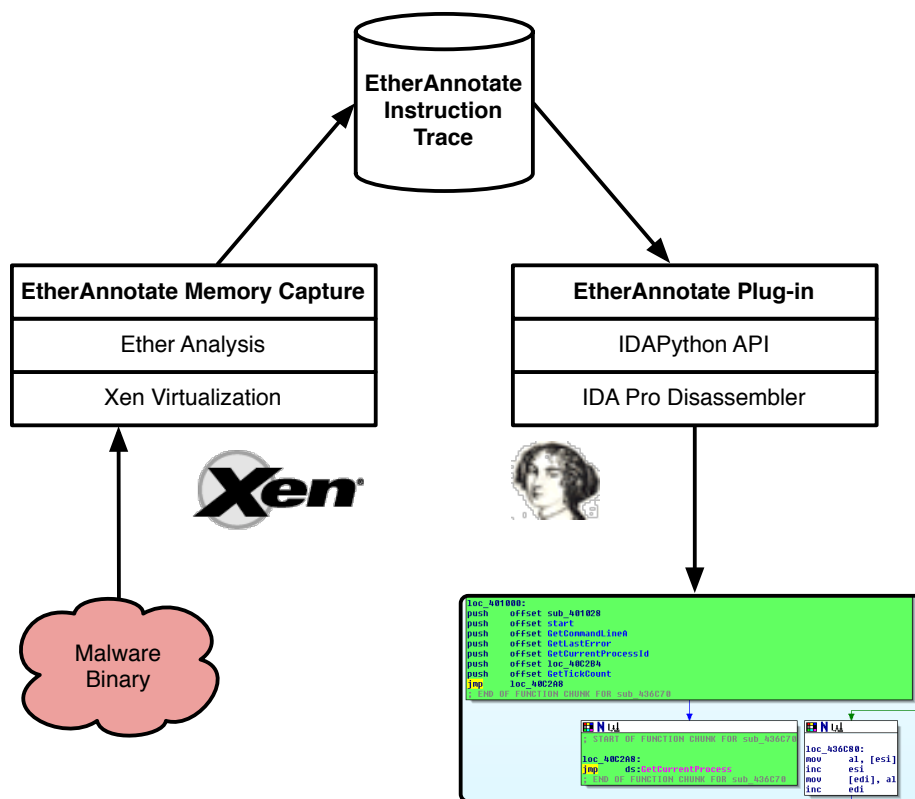


Figure 3.1. The Workflow for Analyzing Malware with EtherAnnotate

Xen Ether, automatic unpackers, or manually - but these techniques are outside the scope of this paper. Once the unpacked version of the binary is loaded and disassembled by IDA Pro, the analyst chooses to load the EtherAnnotate plug-in from IDA's menus and chooses the EtherAnnotate instruction trace file produced by the first task. At this point, IDA will execute the plug-in's code and produce the results detailed in Section 3.3.

3.2. XEN ETHER MEMORY MANIPULATION ANALYSIS

The first half of the EtherAnnotate plug-in consists of modifications to the Xen Ether malware analysis framework in order to record additional data about the malicious program's state during execution. Specifically, the plug-in provides logged

access to the run-time values of all referenced registers and uses a simple heuristic to also record all potential string array references by pointers.

3.2.1. Hardware and Software for Development. A computer built with an Intel Core2Duo E6400 processor (with Intel VT-x) and 1GB of DDR2 RAM was used during both development of EtherAnnotate and the further analysis provided in Section 4. The Debian Lenny 5.0 Linux distribution was installed as the Dom0 operating system running on top of Xen 3.1. Finally, Xen Ether 0.1 was installed via the official instructions² with a Windows XP SP2 DomU guest used for analysis which was allocated 256MB of RAM.

3.2.2. Retrieving Instruction Trace from Malware Sample. The goal of the Xen portion of EtherAnnotate is to single step a single virtualized guest program and record the register values during each instruction. In order to enable single stepping of a processor, the x86 architecture provides a single bit Trap Flag (TF) that can be set in the FLAGS register. The FLAGS register is a status register on x86 processors that contains various bit-flags that control or describe the state of the processor. If the Trap Flag is set to true then the processor will make a call to interrupt 1 (INT 1) after each instruction is executed. This interrupt is typically used by kernel debuggers which would set this flag on the physical processor when running in a bare-metal environment. In the Xen virtualization environment however, the hypervisor creates a Virtual-Machine Control Structure (VMCS) for each virtual processor of each virtual machine. The VMCS is a structure defined in the Intel VT-x documentation that the hardware virtualization extensions use to store the state of each virtual processor. In addition to virtual registers and flags that the hardware processor has, the VMCS contains VM-specific control fields to set which operations should trap into the hypervisor, where the processor should start executing during a VM-exit exception, and metadata about the cause of a VM-exit [25]. The Xen hypervisor can trap on specific interrupts and has been modified by Ether to allow analysis code to be run while the virtual processor is paused at each instruction.

The process that Ether uses to enable single stepping of a virtual guest is presented in Figure 3.2. The code flow originates in the original Xen hypervisor, specifically the call to *vmx_properly_set_trap_flag()* which is normally used by Xen to properly

²<http://ether.gtisc.gatech.edu/source.html> (released April, 2009)

handle returning execution to the guest during a VM-enter. If single stepping was selected during the Ether configuration, then the original Xen code diverges into a section which sets the Trap Flag for the virtual processor and also sets the VMCS exception bitmap such that the Xen hypervisor will trap on the next exception. This causes the guest to return, execute a single instruction which triggers an INT 1 exception due to the Trap Flag being set, and the Xen hypervisor catches this exception and sends code execution to *vmx_handle_debug_exception()*. Still inside the Xen hypervisor (code modified by Ether), the code receives the debug exception (which signifies interrupt 1 was triggered) and if Ether has enabled single stepping then it makes a call to the next Xen Ether hypervisor function, *ether_handle_instruction()*. It is in this instruction handler that the Xen hypervisor finally gives control to the Ether Dom0 code by sending an ETHER_NOTIFY_INSTRUCTION message to the Dom0 code via a shared memory page. In the Ether main loop (and code section where EtherAnnotate modifications were made), the program awaits messages sent from the hypervisor - when an ETHER_NOTIFY_INSTRUCTION message arrives, it calls a final function named *disasm_instruction()* which handles the disassembly of a virtual guest's current instruction.

The previous actions were required in order to force the virtual guest operating system to essentially pause after each instruction it executes, allowing the EtherAnnotate code to execute and analyze the virtual guest's state while it is paused. Figure 3.3 continues where Figure 3.2 ended and describes the data and code flow used to parse individual machine instructions and capture the register and memory values. First, an additional step is required before *vmx_handle_debug_exception()* is called; the EtherAnnotate algorithm does not need to see all instructions executed on a virtual processor but is only interested in viewing the instructions executed by a single process within the virtual guest. When the user executes the EtherAnnotate analysis program, one of the command arguments is the malware filename, N_m , which the user wishes to filter by. Every operating system uses its own method for loading a process into memory and executing it and fortunately the Ether developers determined how Windows XP SP2 loads processes and can determine the current executing process by filename. Once Ether detects that Windows has switched to a process whose filename matches the filter provided by the user, Ether will determine

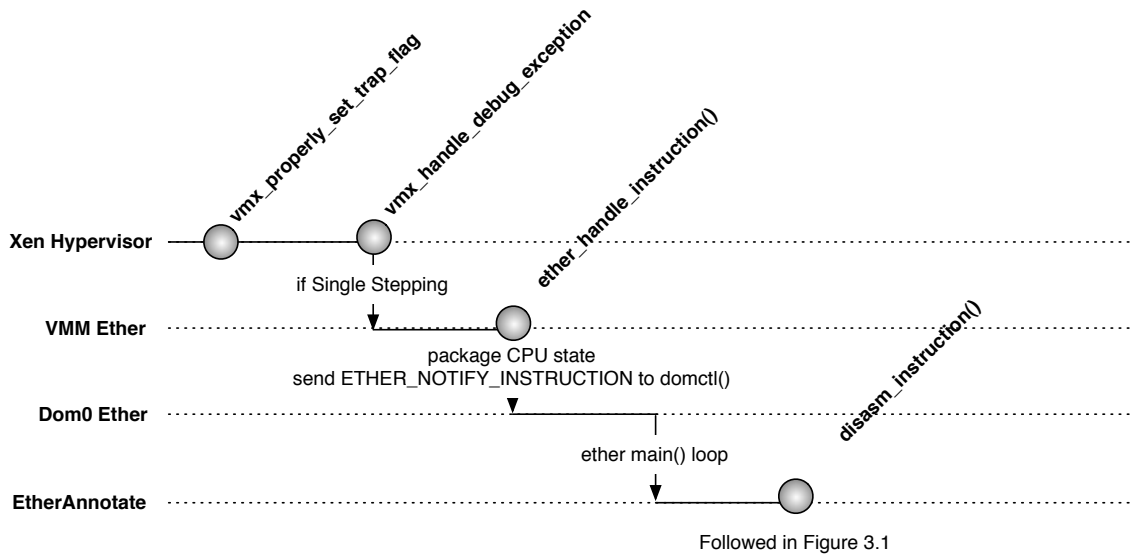


Figure 3.2. Control and Data flow during Ether Single Stepping

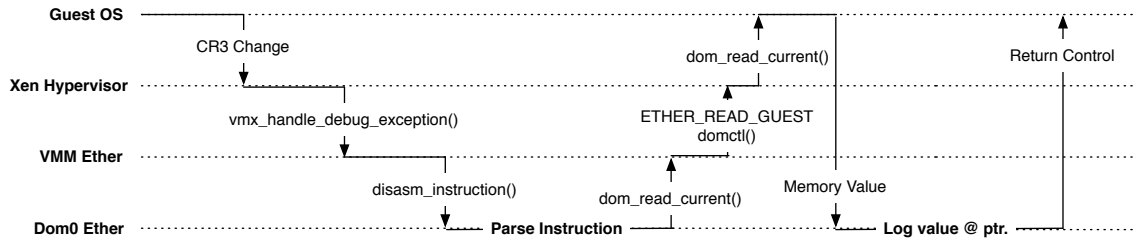


Figure 3.3. Control and Data flow during an EtherAnnotate Call

the value of Control Register 3 (CR3), $CR3_m$ and instruct the hypervisor to trap whenever CR3's value is changed to $CR3_m$. This is useful since CR3 is a processor register (virtualized to the guest) used for virtual memory addressing and always contains the page directory base register (PDBR) which is unique for each process on an operating system [26]. Therefore, if another process P_x with the same filename as the malware ($N_x = N_m$) is executed after the malicious program P_m has started then Ether will only single step execution for P_m since $\forall x, CR3_x \neq CR3_m$.

At this point, Ether has enabled single stepping for a specific malware process based on a filename argument provided by the end-user and the guest is paused for analysis. The Ether code originally called *disasm_instruction()* which would use `libdisasm`³ to parse the x86 opcode into a human-readable string and log the disassembled instruction to a file. EtherAnnotate modifies this section of code to also record the values of all registers which appear in the operands of the instruction and also follow pointer values to possible strings in memory. Algorithm 1 describes the process that EtherAnnotate uses to log instruction register values and the potential string arrays that the reference values point to in memory. Ether already uses `libdisasm`'s *x86_disasm()* function to generate a structure containing all information about each instruction, so Algorithm 1 begins after this call. For each trapped instruction of the guest operating system, EtherAnnotate first uses the `libdisasm` library to parse the instruction's operation code (opcode, the raw binary language a processor interprets) and determine if one of the x86 general registers is used - EAX, EBX, ECX, EDX, ESP, EBP, EDI, or ESI. Next, for each operand o_m that contains a register value, the value of that register is stored into *val* by accessing internal Xen state structures for the virtual machine under analysis. If *val* is of non-zero value, then the code proceeds in an attempt to treat o_m as a pointer and find the value of the memory it references. To do this, a call is made to the internal Xen function *domain_read_current(val)* which does a mapping from the virtual machine's virtual memory into the host machine's virtual memory and returns the value stored at the address *val*. Next, the algorithm uses a basic heuristic to determine if the register in o_m is pointing to a string array (run-time strings will be useful in analysis) by searching for ASCII characters. The current value in memory that the pointer points to is checked to see if it is within the printable ASCII characters (0x20 to 0x7F); if it is, then the value is copied into a buffer, the address is incremented by one byte, and the loop repeats. Once either a predefined maximum number of characters have been copied from the string or a non-ASCII character is reached then the loop completes and all processing for the current instruction is finished other than logging the results out to the EtherAnnotate instruction trace file.

³<http://bastard.sourceforge.net/libdisasm.html>

ALGORITHM 1 Register and C String Array Logging

```

var: string[MAX_SIZE] {Limited to 32 bytes in prototype}
var:  $\forall i, reg_i \in \Sigma$ 
var:  $\Lambda =$  all printable ASCII characters
for instruction  $I_m$  in  $P_m$  do
  for operand  $o_m$  in  $I_m$  do
    if  $o_m \in \Sigma$  then
       $val = xen\_state.register\_values[o_m]$  {Store value of all registers used in
      operands}
      if  $val \neq 0$  then
         $ptr\_val = domain\_read\_current(val)$  {If value is not null, assume it's a
        pointer and load memory at value}
         $j = 1$ 
        while  $ptr\_val[j] \in \Lambda$  and  $j < MAX\_SIZE$  do
           $string[j] = ptr\_val[j]$  {If memory value contains ASCII characters,
          store them in a buffer}
           $j = j + 1$ 
        end while
      end if
    end if
  end for
  print  $I_m : (\forall i, o_m) : (\forall j, ptr\_val_j : string[j])$  {Log all register values and their
  possible string references for each instruction}
end for

```

The worst-case runtime complexity of this algorithm is $O(IRS)$ where I is the total number of instructions that are traced, R is the average number of operands containing register values per instruction, and S is the maximum string length constraint set by the analyst. Ether already requires $O(I)$ time to single-step and perform an instruction trace and EtherAnnotate adds to Ether's instruction trace runtime by $O(RS)$. The x86 instruction set only contains instructions with a possible zero to three operands, so in the worst-case scenario all instructions will contain three operands - all of which contain register values (extremely unlikely). Therefore, the worst-case runtime complexity that EtherAnnotate adds to the previous Ether instruction trace operation is $O(3S)$. Realistically, the call to `domain_read_current()` takes the most amount of additional time since it requires a hypercall into the Xen hypervisor and must wait for data to be returned.

Table 3.1. Performance Analysis Between Ether Tracing and EtherAnnotate Tracing

	Ether Size	EA Size	Ether Time	EA Time
calc.exe	932 KB	2,221 KB	260.4s	253.8s
iexpress.exe	804 KB	2,003 KB	84.3s	84.9s
winmine.exe	364 KB	815 KB	64.3s	65.2s
notepad.exe	146 KB	414 KB	73.1s	76.2s
ipconfig.exe	262 KB	656 KB	111.4s	117.8s
SpyEye.exe	127,593 KB	347,358 KB	268.2	296.4

To examine the experimental performance of EtherAnnotate’s additions to the Xen Ether code, multiple binaries were traced using each system and the amount of time taken was recorded. Since Xen EtherAnnotate’s instruction tracer runs independently from the virtual machine (can not induce programs to be executed inside of the guest), the best method for recording the time taken was to start the timer with the launch of EtherAnnotate then manually start the program to be traced in the guest. This timing method is not particularly accurate, but should can at least give an estimate to the amount of overhead incurred by the EtherAnnotate code additions. Since each timing had to be completed manually, each program was just run once with the timer for each condition - tracing with Ether and tracing with EtherAnnotate. Additionally, the size of the trace files created by both Ether and EtherAnnotate are compared in order to present the storage overhead of using EtherAnnotate. The timing and storage comparisons are shown in Table 3.1. The percentage increases incurred on the storage and time requirements for using EtherAnnotate are displayed in Table 3.2. Finally, the percentage increases are visualized in Figure 3.4 as well as the best linear fit equation for the data.

3.2.3. Retrieving Memory Values During Execution. For each operand that contains a general purpose register, the EtherAnnotate code parses Xen’s internal structure for the virtual machine under analysis and stores the current value of the register used in the operand. If the register’s value is 0, then no further action is taken for that specific operand (this is a simple heuristic to determine if a register has a useful value). The algorithm next assumes that whatever value the register holds could potentially be a pointer to a string array. A call to *domain_read_current(val)* is

Table 3.2. Percentage Difference in Time and Speed of Ether and EtherAnnotate

	EA % Additional Size	EA % Additional Time
calc.exe	+138.3%	-2.53%
iexpress.exe	+149.1%	+0.71%
winmine.exe	+123.9%	+1.40%
notepad.exe	+183.6%	+4.24%
ipconfig.exe	+150.4%	+5.75%
SpyEye.exe	+172.2%	+10.51%
Average:	+152.9%	+3.35%

made which is an internal Xen function that maps a guest domain's virtual memory address into the host machine's physical memory address and allows EtherAnnotate to read memory at a specific location in the guest. If register holds a pointer value to a location in the guest's memory which contains printable ASCII characters, then the string array is copied out of memory into a buffer until either a non-printable ASCII character is reached or 32 bytes have been copied (the static size limit was an implementation limitation and not a limitation of the methodology). Finally, the instruction mnemonic, the values of registers referenced in the instruction operands, and any possible string arrays pointed to are logged into a text file for each instruction in the malware's trace.

3.3. IDA PRO ANNOTATION PLUGIN

While logging all register values and potential string values of a malicious program during its runtime may be useful for analysis, a graphical representation of this data would provide a significantly more useful tool to the malware analyst. The EtherAnnotate IDA Pro plug-in provides a simple method for loading a custom EtherAnnotate instruction trace and annotating disassembled instructions with their runtime values in a meaningful graphical representation. The IDA Pro Disassembler⁴ is one of the most popular and robust software disassembler tools available to malware analysts. In addition to providing a disassembler for many common processor architectures, the IDA Pro application has a Software Development Kit (SDK) that can be used to build plug-ins that are able to access and manipulate many of the internal

⁴<http://www.hex-rays.com/idapro/> (last updated December, 2009)

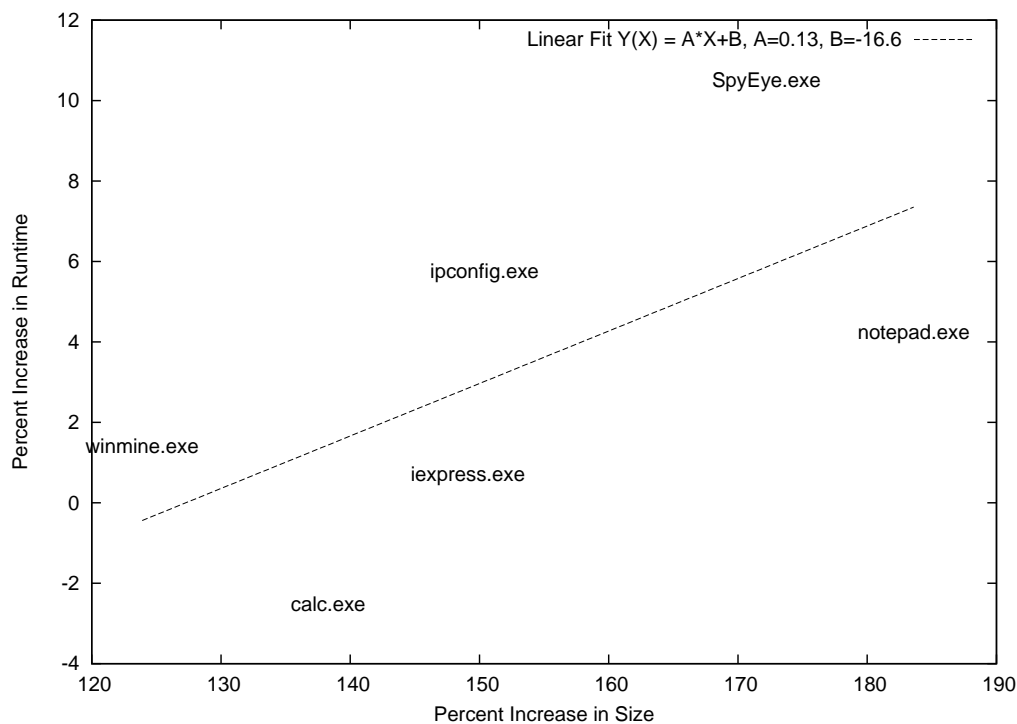


Figure 3.4. Correlation of the Percent Increase of Filesize and Runtime using EtherAnnotate

and GUI aspects of IDA Pro. The SDK provided by the original developers requires the plug-in developer to write their plug-in in the C language and recompile their code for each revision.

3.3.1. IDA Python Development Environment. During EtherAnnotate's development, the formatting and information stored in the log files from Xen Ether changed as new design ideas emerged and thus was more pragmatic to use an SDK better suited for rapid, agile development. Fortunately, a developer for the F-Secure security company named Gergely Erdélyi published an extension of the official IDA Pro SDK, named IDAPython⁵, which uses the Python language and provides all

⁵<http://d-dome.net/idapython/> (last updated July, 2009)

of the original functionality of the IDA Pro SDK in a scripting language which favors rapid prototyping.

3.3.2. Enhancements to IDA Disassemblies. The goal of EtherAnnotate's IDA Pro enhancements was to provide malware analysts with strategically targeted run-time register and memory values in a easily comprehensible graphical addition. The first half of this research's approach creates comment annotations for all disassembled instructions which are listed in a corresponding Xen EtherAnnotate trace log. For each instruction listed in the log file from the Xen-side, the IDA Pro plug-in uses regular expression pattern matching to store the instruction mnemonic string, the address in code memory of the instruction, and any annotations for that instruction that were recorded by EtherAnnotate. Then, for each annotation found, a comment is added or amended to the address location in IDA Pro with the contents of the annotation. An example of the annotations taken from a sample malware's decryption routine is presented in Figure 3.5. In this figure, only the potential string array values are displayed and from these it appears that the code is incrementally decoding various strings each time the section is called. The lines of instructions that are highlighted in green are instructions that were executed during the EtherAnnotate trace, the instructions with a white background were not executed during the trace.

The second half of the EtherAnnotate IDA Pro plug-in consists of a simple method for conveying code coverage in IDA's graph layout mode. For each instruction that the malware executed during analysis under Xen EtherAnnotate, the address of that instruction is located in IDA Pro's disassembly and the background of that line is colored such that a quick analysis of the code will make the code flow path obvious. Finally, Algorithm 2 provides an overview of the IDA Pro plug-in's actions taken to annotate all instructions and render colorized code coverage.

The runtime complexity of the annotation algorithm is only $O(IA)$ where I is the total number of instructions in the trace file and A is the average number of annotations per instruction file. Each instruction can have a maximum of three operands (see Section 3.2.2), and in the worst-case each of these operands will also contain a register which points to a string value - giving a maximum of 6 annotations

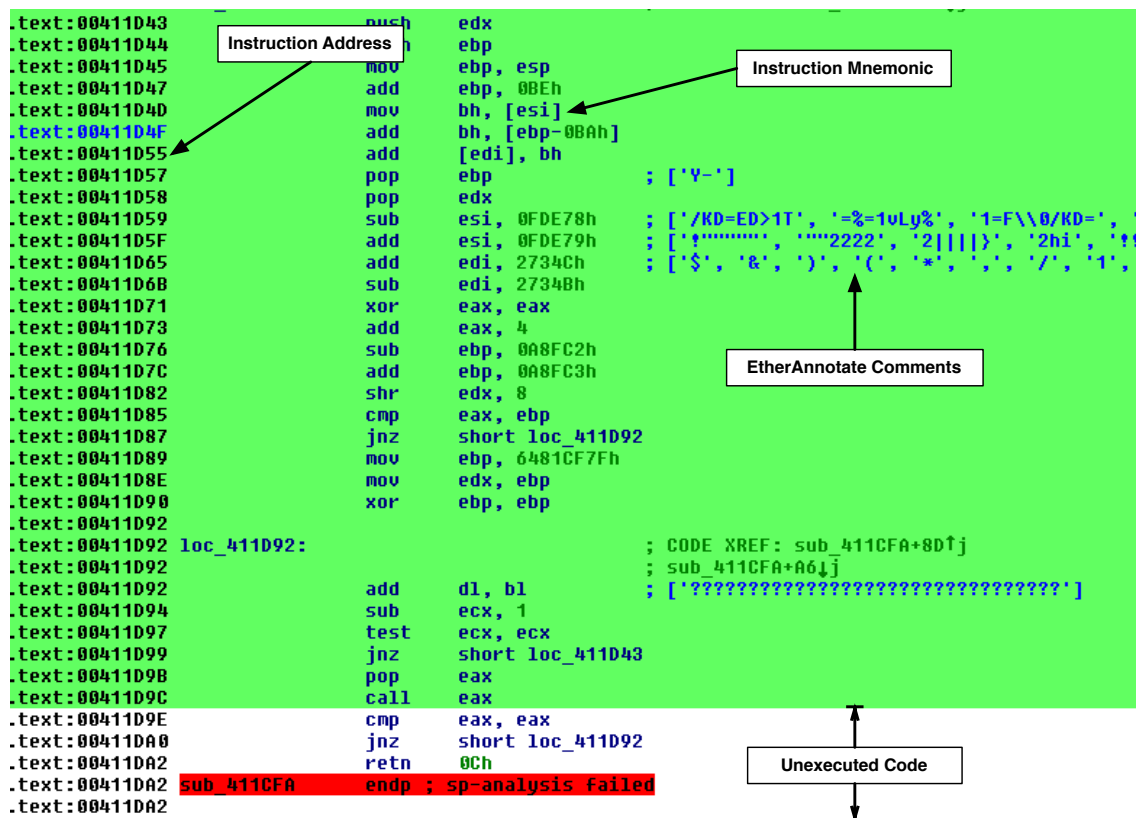


Figure 3.5. Example of Annotated Instructions during a Decryption Routine

per line (3 register values, and 3 string values). Therefore, the worst-case runtime of the annotation algorithm is linear in $O(6I)$.

3.4. IMPROVEMENTS MADE DURING CASE STUDIES

At the start of the second case study of this thesis (Section 5), the author's goal was to find a popular malicious binary that detects and evades common dynamic analysis tools. As it turned out, one of the core challenges in front of this goal was the lack of speed that EtherAnnotate that truly manifested during analysis of complex malware that requires many instructions to execute. This issue had been known on the Xen-side since early development and was caused by an unknown bug with the interaction between EtherAnnotate and Xen. Once the speed bug was fixed as described in Section 3.4.1, it was then apparent that the IDA Pro EtherAnnotate

ALGORITHM 2 Annotation and Code Coverage Visualization

```

trace_file = LOAD(malware.trace)
for Line L in trace_file do
  instruction_string = regular expression match
  instruction_address = regular expression match
  annotations = regular expression match
  SetColor(instruction_address, GREEN)
  for annotationi in annotations do
    MakeComm(instruction_address, annotationi)
  end for
end for

```

plug-in was also too slow for effective malware analysis - the IDA Pro speed increases are described in Section 3.4.2.

3.4.1. Xen EtherAnnotate Speed Increases. From almost the beginning of development EtherAnnotate’s development the tool would analyze the same instructions many times in a row instead of just once each like they were actually executed. The trace file would contain each instruction repeated a variable number of times, but almost always more than once, but the execution of the program in the virtual machine would be consistent with execution while not tracing (instructions were only executed once in the virtual machine). Since EtherAnnotate would cause Xen to trap each instruction multiple times, the execution of EtherAnnotate’s instruction tracing function was many orders of magnitude slower than Ether’s original instruction tracer. The reason of this issue was never discovered until when the author was analyzing malware during the second case study.

Prior to the second case study, the slowness issue was assumed to be caused by the addition of the `domain_read_current()` function to Ether’s original instruction tracer. This was the only function used in EtherAnnotate that called into the internals of Xen and allows EtherAnnotate to retrieve the memory value at the address contained in a register while the virtual machine is paused for each instruction. Since smaller binary files were used during development and testing, a simple workaround was designed which checked if the processor’s instruction pointer (EIP) had changed since the last time Ether received an instruction trace event from Xen. If it was found that the value of EIP had not changed from the previous instruction trace event,

Ether would then return control to Xen and not execute EtherAnnotate's code. This workaround eliminated multiple instructions being recorded into the EtherAnnotate trace files but did not eliminate the extreme speed decrease.

During the second case study, the culprit of this issue was discovered to be the method that `domain_read_current()` uses to read memory from a Xen virtual machine. The `domain_read_current()` function is located in the Ether userland source file `syscalls.c` which simply redirects to another userland function call, `ether_readguest()`. The function `domain_read_current()` is located in the userland source file `ether.c` which sends a *domctl* message containing the address of memory to retrieve, a buffer to store the value in, and the command `XEN_DOMCTL ETHER_READ_GUEST`. A *domctl* message is the system that Xen uses to allow userland applications in Dom0 make command requests to the Xen hypervisor (a hypercall). All of the Xen Ether domctl command handlers are located in the Xen hypervisor source file `/xen/common/domctl.c` in the function `do_domctl(u_domctl)`. This function takes a domctl command and determines what actions should be taken and what values returned to the caller based on a switch table of the domctl command (`XEN_DOMCTL ETHER_READ_GUEST` in this example). The key element of this function that caused Xen to trap on the same instruction multiple times is that before the switch table there is a call to `domain_pause(d)` which pauses the current domain. This call to pause the current domain is understandable since all of the original Xen domctl commands would be used while the virtual machine is running (e.g., to check the status of the virtual CPU). However, at this point in EtherAnnotate's code the virtual machine has already been paused since it is being single stepped through the malicious binary's execution. A second call to `domain_pause(d)` while the virtual machine is already paused has undefined behaviour and in this case appears to cause the virtual machine to not step to its next instruction for a significant amount of time.

In order to alleviate the problem found above, the `/xen/common/domctl.c` Xen hypervisor function was patched so that the virtual machine will not be paused during `XEN_DOMCTL ETHER_READ_GUEST` domctl commands. An example of the runtime speed up that this patch created is the SpyEye trojan used for the second case study. With the original Xen EtherAnnotate code, this instruction trace would take three to four

```

215     if(d->is_dying || d->is_shutting_down || d->is_shut_down)
216     {
217         printk("ETHER: domain %d is dying or dead\n",
218             d->domain_id);
219         break;
220     }
221
222     if(op->u.ether.command_code != XEN_DOMCTL_ETHER_READ_GUEST)
223         domain_pause(d);
224     switch(op->u.ether.command_code)
225     {

```

Figure 3.6. Area of Xen Ether Source Code Changed for Pausing

```

532     default:
533         printk("ETHER: invalid command to domctl\n");
534         ret = -ENOSYS;
535         break;
536     }
537     if(op->u.ether.command_code != XEN_DOMCTL_ETHER_READ_GUEST)
538         domain_unpause(d); /* causes guest to latch new status */
539     rcu_unlock_domain(d);
540 }
541 break;

```

Figure 3.7. Area of Xen Ether Source Code Changed for Unpausing

hours until execution finished. After the patch, the trace would only require around ten minutes until the SpyEye binary had finished execution. Figure 3.6 shows the modifications made to create a conditional pause and Figure 3.7 shows the same modification made for a conditional unpauses.

3.4.2. IDA Pro EtherAnnotate Porting to C++. The original IDA Pro EtherAnnotate plug-in described in Section 3.3 was developed in the IDAPython API which is a wrapper API for the official IDA Pro C++ API. This decision made sense during development since the trace file format was changing as the Xen-side development progressed and Python provided native libraries for regular expressions which were used to parse the input trace files. However, once larger traces were now practical produce with the significantly faster Xen EtherAnnotate tool, these trace files were too large to be processed in a timely manner with the IDAPython plug-in.

For example, an EtherAnnotate trace of the Conficker.C worm was over 10GB and only took a matter of minutes to generate with Xen EtherAnnotate. However, a similar trace file from the SpyEye trojan was 700MB and did not finish after being run for over 12 hours with the IDAPython plug-in. Clearly, processing a 10GB would take a significantly longer (although linearly) amount of time when a 700MB trace taking 12 hours is already too long to be practical for analysis.

The Python language is an interpreted language which is inherently slower than compiled languages such as C or C++. Additionally, the IDAPython library is a wrapper around the IDA Pro C++ API which means that an API call to the IDAPython library must be parsed by IDAPython first and then sent to the native C++ API. Because of these two traits of IDAPython, it was decided that porting the EtherAnnotate plug-in to the native C++ API would be the best tactic for increasing processing speeds. A direct port of Algorithm 2 was first attempted using the Boost⁶ regular expression library for C++ to accommodate the regular expression portions of the original Python plug-in. As the port developed, it was discovered that including the Boost regular expression matching calls significantly increased the runtime of the C++ plug-in and even sometimes eclipsed the Python plug-in's runtime. Therefore, the Boost library regular expression calls were removed and the algorithm was optimized to only use the C++ native string functions to parse each line of the trace log file. The final C++ plug-in runtimes for parsing EtherAnnotate instruction trace files was found to be significantly faster than the original IDAPython plug-in. The previous example of the SpyEye Trojan's 700MB trace which did not complete execution after 12 hours now finishes in under 10 minutes.

⁶<http://www.boost.org/> (last updated February, 2010)

4. CASE STUDY OF MEGAD BOTNET DROPPER

This section presents a case study performed to analyze a malicious botnet binary file with the new EtherAnnotate tool developed in this thesis. Section 4.1 gives a detailed analysis of the malware following the same steps as during the actual case study. Section 4.2 reports on the findings of the analysis, including the actions that the malware takes, the goals of each section of the code, and how the annotated instruction traces and code coverage indicators helped analysis.

4.1. MALWARE ANALYSIS USING ANNOTATED TRACING

The analysis performed in this section is analogous to what a computer security lab would complete when analyzing newly seen malware. A sample malware binary labeled as the MegaD botnet⁷ was downloaded from the Offensive Computing⁸ repository which stores live malware that other users have found. Once the sample was retrieved, it was loaded onto a Windows XP SP2 Xen image with the EtherAnnotate framework installed. The sample was executed once with the annotated instruction tracing tool recording all instructions that were executed and all register values during the execution. Afterward, the Xen image was restored to a clean state and the malware was again loaded onto the image. The sample was executed a second time while the Ether automatic unpacking tool monitored and stored any layers of unpacking that appeared.

After running the MegaD sample through the annotated instruction trace tool and the Ether unpacking tool, an instruction trace file and a single unpacked layer binary were now available to help static analysis. The unpacked binary can be loaded into the IDA Pro disassembler and the IDA Python EtherAnnotate plug-in will annotate all instructions using the trace file previously generated as shown earlier in Figure 3.5.

4.1.1. Part I - Loading Functions. After loading the annotated binary into IDA, it appears that the first part of the code is a large loop from visual inspection of the code coverage in Figure 4.1. Further inspection of the actual instruction

⁷MD5: 5AFEB6643C8E1A83A3B32908F3921829

⁸<http://offensivecomputing.net/> (accessed March, 2010)

trace confirms that the first 99.9% of the instructions executed are related to this loop (most likely a code decryption routine which is decrypting code for the final 0.1% of instructions executed). It was discovered during this case study that either IDA Python or the annotation plug-in cannot handle extremely large trace files (this particular file contains over 2.7 million executed instructions) so the trace had to be divided into the last 0.1% in order to properly annotate the rest of the code. The large box at the bottom of Figure 4.1 is where the code jumps to after it has finished the main loop; Figure 4.2 presents the final lines of the loop in more detail. Further analysis using OllyDbg helped determine that this initial decryption loop is part of the publicly available Ultimate Packer for eXecutables (UPX) program which compresses and obfuscates binary executables⁹.

So far, the code coverage and instruction trace have been used to quickly find that the MegaD binary executes a relatively small loop a very large number of times and finally jumps to a separate section of code once it has finished. At the current analysis location in Figure 4.2, it appears that the code pushes seven sets of addresses to the stack and then jumps to another location in memory (an import address of a system call in the import table). After further analysis of this particular binary, it turns out that this is a common method for calling a new function, possibly to obfuscate the call, and is described in more detail in Section 4.2.2.

Since the Ether unpacking tool does a raw dump of a process from memory as it is executing, it does not obtain a properly formatted PE executable formatted binary. One side-effect of this is that the table of imported system functions contained inside of every PE file is not properly referenced or formatted. While IDA Pro will usually annotate imported function names with their arguments (which helps the malware analyst), it is necessary to manually rename imported functions to their Windows system call names in IDA when using an Ether dump. The names associated with function addresses can be found online¹⁰ and an example of how IDA appears before and after are presented in Figures 4.3 and 4.4 respectively.

Once all of the initial clean-up and analysis is done in IDA Pro, the analysis process can start to look at the decrypted functions that are now run by the malware

⁹<http://upx.sourceforge.net/> (last updated September, 2009)

¹⁰<http://www.newsvoter.com/Binary/dll/index.html> (accessed March, 2010)

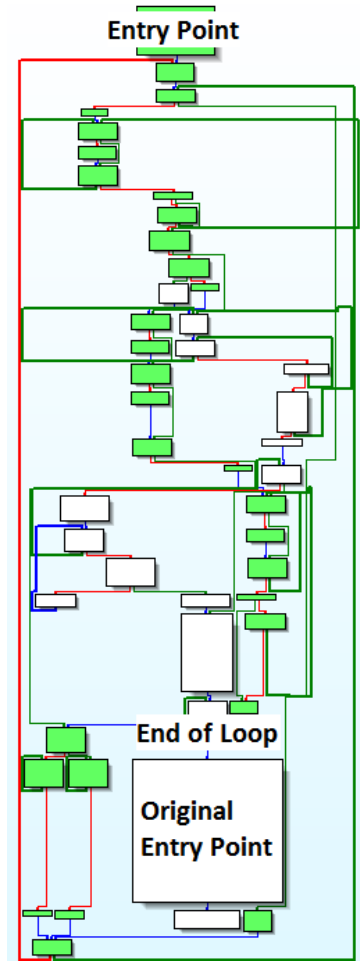


Figure 4.1. The initial decryption loop of MegaD

```

loc_401000:
push  offset sub_401028
push  offset start
push  offset GetCommandLineA
push  offset GetLastError
push  offset GetCurrentProcessId
push  offset loc_40C2B4
push  offset GetTickCount
jmp   loc_40C2A8
; END OF FUNCTION CHUNK FOR sub_436C70

; START OF FUNCTION CHUNK FOR sub_436C70
loc_40C2A8:
jmp   ds:GetCurrentProcess
; END OF FUNCTION CHUNK FOR sub_436C70

loc_436C80:
mov   al, [esi]
inc   esi
mov   [edi], al
inc   edi

```

Figure 4.2. Final lines of decryption routine and jump to malicious code

```

• UPX1:0042C190 dword_42C190 dd 7C830053h ; DATA XREF: sub_40C266Tr
• UPX1:0042C194 dword_42C194 dd 7C810F9Fh ; DATA XREF: sub_40C34ATr
• UPX1:0042C198 dword_42C198 dd 7C812851h ; DATA XREF: sub_40C2F6Tr
• UPX1:0042C19C align 10h
• UPX1:0042C1A0 dword_42C1A0 dd 77D65D61h ; DATA XREF: sub_411727+166
• UPX1:0042C1A4 dword_42C1A4 dd 77D48697h ; DATA XREF: sub_411727+93C
• UPX1:0042C1A8 dd 0
• UPX1:0042C1AC dword_42C1AC dd 71AB3EA1h ; DATA XREF: sub_40C3FETr
UPX1:0042C1AC ; sub_40C79B+30Tr ...
• UPX1:0042C1B0 dword_42C1B0 dd 71AB2C69h ; DATA XREF: sub_40C3F8Tr
UPX1:0042C1B0 ; sub_424422+E2Tr
• UPX1:0042C1B4 dword_42C1B4 dd 71AB428Ah ; DATA XREF: sub_40C3F2Tr
UPX1:0042C1B4 ; sub_40C79B+191Tr ...
• UPX1:0042C1B8 dword_42C1B8 dd 71AB2DC0h ; DATA XREF: sub_40C3ECTr
UPX1:0042C1B8 ; sub_41C112+1685Tr ...
• UPX1:0042C1BC dword_42C1BC dd 71AB2D0Fh ; DATA XREF: sub_40C3E6Tr
UPX1:0042C1BC ; sub_424079+60Tr
• UPX1:0042C1C0 dword_42C1C0 dd 71AB615Ah ; DATA XREF: sub_40C3E0Tr
UPX1:0042C1C0 ; sub_40C79B+198Tr ...
• UPX1:0042C1C4 dword_42C1C4 dd 71AB88D3h ; DATA XREF: sub_40C3DATr
• UPX1:0042C1C8 dword_42C1C8 dd 71AB4519h ; DATA XREF: sub_40C3D4Tr
UPX1:0042C1C8 ; sub_40D15F+1E6Tr ...
• UPX1:0042C1CC dword_42C1CC dd 71AB2BF4h ; DATA XREF: UPX0:0041F90DTr
UPX1:0042C1CC ; UPX1:00420824Tr
• UPX1:0042C1D0 dword_42C1D0 dd 71AB2B66h ; DATA XREF: sub_40C79B+68Tr
UPX1:0042C1D0 ; sub_40D8EC+313Tr ...
• UPX1:0042C1D4 dword_42C1D4 dd 71AB2BC0h ; DATA XREF: sub_40C79B+160
UPX1:0042C1D4 ; sub_40C79B+1DDTr ...
• UPX1:0042C1D8 dword_42C1D8 dd 71AB46C9h ; DATA XREF: sub_40C3CETr
UPX1:0042C1D8 ; sub_41C112+1814Tr

```

Figure 4.3. Example of the Unpacked Import Table

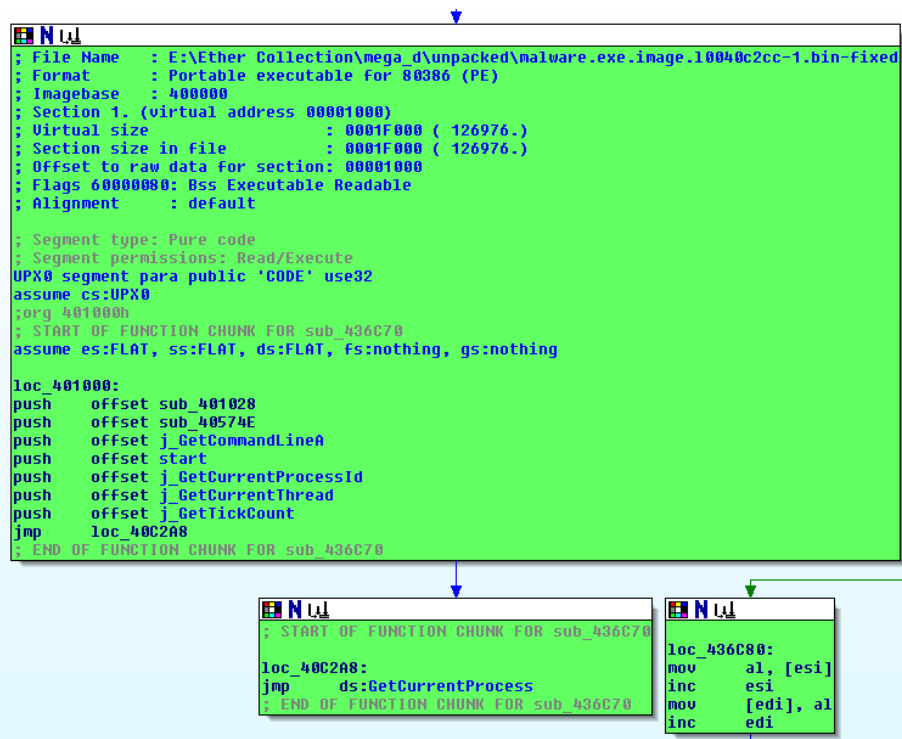


Figure 4.4. Jump Sequence after Imports have been Fixed

and see if the annotations from EtherAnnotate assist in the investigation. The first action that the MegaD binary takes after decryption is to call the Windows system function `GetVersionEx` which returns operating system version information into an `OSVERSIONINFO` struct¹¹. This is a fairly common method that malware uses to determine what version of the Windows OS it is running on (9x, XP, or Vista/7 mostly) and is needed in order to properly load some functions and DLLs later. The next section of code is displayed in Figure 4.5 and uses various MMX registers and MMX-based instructions to decrypt and store the string “kernel32.dll” into the EDX register and push it onto the stack. First, the MMX instruction set is an extension to the x86 instruction set and provides registers and instructions designed to efficiently handle certain floating point operations [26]. The code here appears to be converting integer values from the standard registers into floating point values, manipulating them, and then converting back to integer values in the standard registers. Second, the annotations provided in this code section make it much more obvious that the goal of this sequence of instructions is to stealthily load a known DLL’s filename out of memory. A non-malicious program would have typically done a simple direct load of a character string’s address which would have been easily traceable in a debugger (which is not what this malware author wanted). Finally, like the jump sequence mentioned above, this pattern of loading a DLL filename string out of memory using floating point manipulation will occur throughout the rest of this part of the malware.

After the first code section, further analysis of the next few sections shows that the code follows a pattern of loading function name strings and calling `GetProcAddress`¹². `GetProcAddress` is another Windows system function commonly seen in malware along with `LoadLibrary`¹³; it retrieves the address of an exported function from a DLL and returns the address. Figure 4.6 shows a zoomed out graph view of the rest of this part of code - each green code block contains MMX-based code to load a function name string, call `GetProcAddress`, and continue to the next block to the right. Again, the annotations in each code block provide the exact function name being loaded that would otherwise be obfuscated by the floating point operations.

¹¹[http://msdn.microsoft.com/en-us/library/ms724451\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms724451(VS.85).aspx) (accessed Mar., 2010)

¹²[http://msdn.microsoft.com/en-us/library/ms683212\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms683212(VS.85).aspx) (accessed Mar., 2010)

¹³[http://msdn.microsoft.com/en-us/library/ms684175\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms684175(VS.85).aspx) (accessed Mar., 2010)

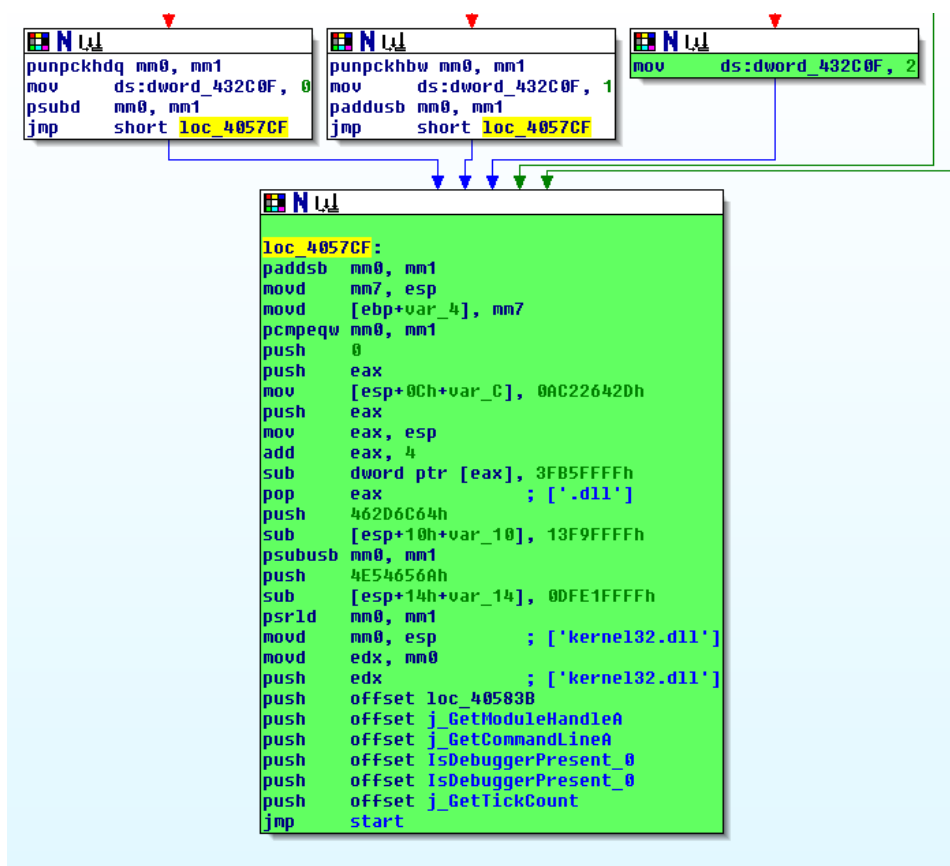


Figure 4.5. Using MMX Registers and Instructions to Load Strings

After all of the needed functions have been loaded with the above technique, the malware finally jumps to a new section, performs a few checks, and finally pushes ResumeThread onto the stack to be called on return and then returns. It appears that the code has potentially been injected into a system application, paused the system application's actual code flow, loaded its own functionality into the system application, and resume execution of the host application. This is a common technique of malware and is similar to parasitic relationships among organic creatures where the malicious code hides in the host application and subtly manipulates its actions.

4.1.2. Part II - Decryption and File Dropping. The final section of the MegaD malware decrypts a section of memory and drops the decrypted data into a .bat file for execution. The first goal of this section is to retrieve the current path

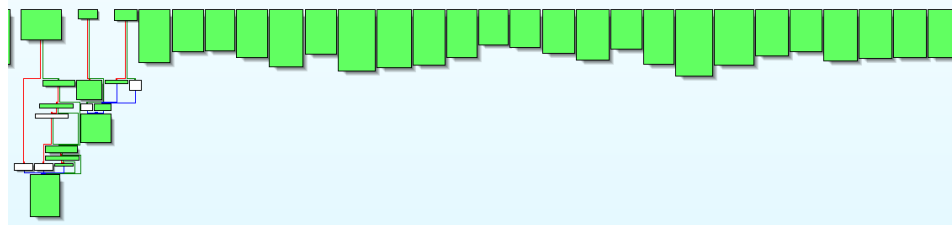


Figure 4.6. Load all needed DLL exported functions.

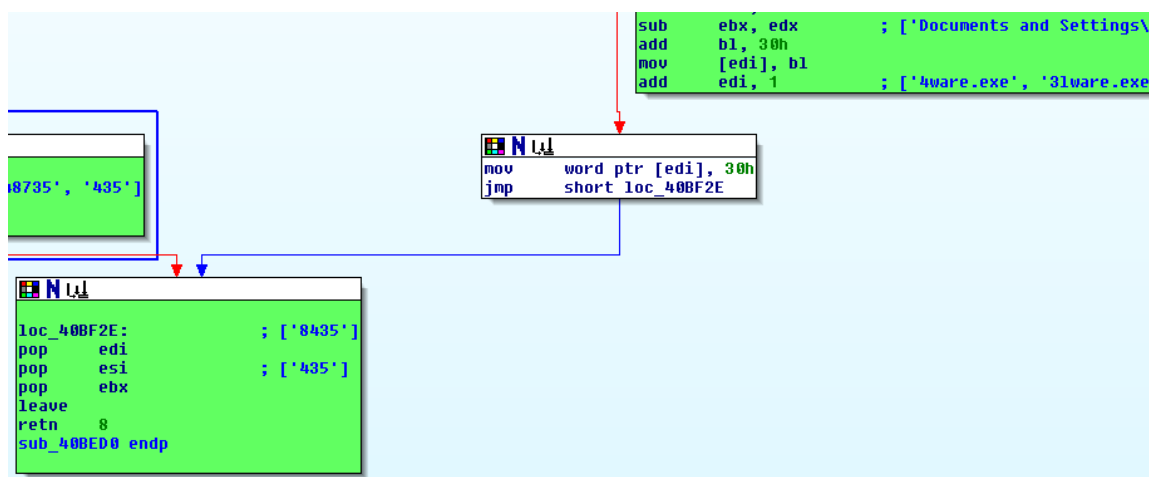


Figure 4.7. Portion of code generating the BAT filename from the current process's filename

of the running application, which the malware achieves by calling the `GetSystemDirectoryA` function (again, this is only known due to the annotation of the string as it is stored in a register). Next, the code jumps around without calling too many functions but appears to be allocating memory on the heap using the `VirtualAlloc` Windows system call - this will be useful when combined with the actions found further in the code. Eventually, both the filename and path of the current process is determined and put into the canonical Windows path format (no slashes, colons, or other symbols omitted). The annotations are particularly useful in the function shown in Figure 4.7 where it appears that the filename of the original program is used to generate the filename of the `.bat` file that will be dropped.



```

loc_402B5A:
psllw  mm0, mm1
push   0           ; hTemplateFile
push   0           ; dwFlagsAndAttributes
push   2           ; dwCreationDisposition
push   0           ; lpSecurityAttributes
push   0           ; dwShareMode
push   40000000h   ; dwDesiredAccess
push   [ebp+var_3C] ; lpFileName
call   j_CreateFileA
punpckhwd mm0, mm1

```

Figure 4.8. The arguments being passed to CreateFileA. Arithmetic operations on register values is currently not supported.

The final section of this malware consists of a long sequence of instructions, starting with a CreateFile function call followed by many WriteFile calls and ending with a CloseHandle call. The CreateFile function call in Figure 4.8 demonstrates how the current implementation of EtherAnnotate lacks the ability to parse certain register values. The final PUSH instruction uses arithmetic operations on a register value before being referenced; the annotation code cannot currently parse this operation and therefore there is no annotation for this line. It would have been quite useful to know the filename being written to, along with other variables throughout the code, but it can be deduced from the previous sections that it will be a .bat filename generated in the previous section. The WriteFile calls write multiple data buffers into the file, followed by DOS batch commands such as “goto”, “if exists”, and “del”. Without further analysis outside of the IDA Pro disassembly, the author was not able to determine exactly what this batch file would perform on execution. After the file handle has been closed, the malware calls ExitProcess and the instruction trace ends.

4.2. REVIEW OF FINDINGS

From the analysis in the previous section, it was determined that this malicious binary’s goal is to inject itself into another executing process, inconspicuously load required DLLs and functions, create a new file based on some sort of hash of the original filename, and write decrypted binary data into this batch file. Much of this investigation’s results come from standard reverse engineering analysis techniques,

but the inclusion of runtime register values in the annotations made some of the more complicated analysis much easier.

4.2.1. Benefits of Annotation and Code Coverage. The benefits of adding code coverage to the EtherAnnotate IDA Pro plugin are the most immediately realized, especially in this example. At the beginning of the case study in Section 4.1.1, it was found that the malware checks which version of Windows it is running on and diverts into one of three code paths from there on out. As it turns out, it can be determined by visual analysis of which code sections have been colored green that the vast majority of instructions in this binary were not executed during the EtherAnnotate trace. Therefore, much of the code disassembled by IDA Pro can be ignored during a quick analysis since it is visually indicated that the program never executed those instructions.

The advantages of having register-value annotations in IDA Pro were touched on during the previous section's analysis. In this example with MegaD, the best example was how the annotation captured the function and DLL name strings that were being loaded through obfuscated means. Without the annotation, these values would have been extremely hard to determine since they were not stored in plaintext in the binary. An alternative method to figure out the values would be to run dynamic analysis tools on the binary (such as ProcMon or Process Explorer) which can determine what exported DLL functions a program loads during runtime. However, these dynamic analysis tools only help determine values that are used during a system call while the program is executing, whereas the EtherAnnotate plug-in captures all string values of registers during any instruction. While testing EtherAnnotate with other malicious binaries, decryption of botnet messages and connection commands have been noticed in the annotations which could aid reverse engineering the protocol of new botnets.

4.2.2. The Mysterious Jump Signature. Before touching on the limits of the current EtherAnnotate tool, a short analysis of the jump signature first seen in Figure 4.2 may provide more insight on how the malware obfuscates its code. Figure 4.9 provides an example of what a standard *call* and *return* instruction sequence typically look like in C-compiled programs. Typically, function arguments are pushed onto the stack prior to a function call; when the instruction pointer jumps to the function code, it accesses these values on the stack as its arguments. Likewise, a

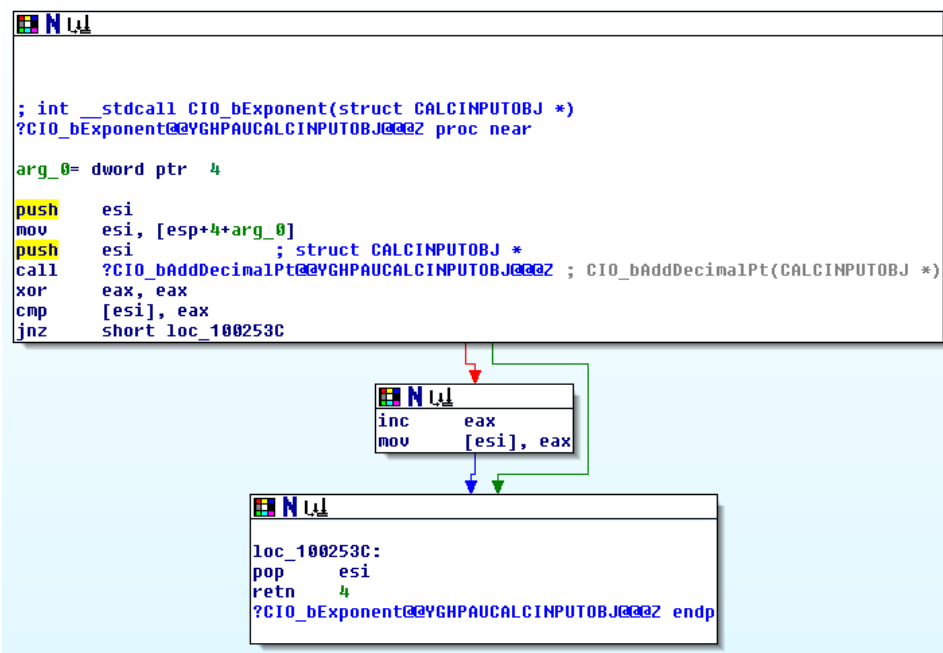


Figure 4.9. Example of a call instruction and return instruction in calc.exe

function typically terminates and returns code flow back to its caller by executing a *retn* instruction. The processor determines where to set the instruction pointer after the *retn* statement by checking the top value on the stack, or the address in ESP [33]. However, in the MegaD code, almost all functional blocks of instructions have the jump sequence in Figure 4.2 where a number of pushes are made followed by a single jump instruction.

It turns out that this obfuscation technique first jumps to the final location specified in the actual *jmp* instruction and then uses the mechanisms of the *retn* instruction to sequentially call all of the other values pushed onto the stack. When each call to each pushed address reaches its final instruction and is ready to return, the return value on the stack is always the previously pushed address in the original code section. For example, in Figure 4.2 the sequence of calls made is GetCurrentProcess (initial *jmp*), GetTickCount, 0x40C2B4 (jumps to GetCurrentThread), GetLastError, GetCommandLineA, and finally an address labeled “start” during prior analysis. Since the results of each call are not pushed to the stack before calling the next

function, it appears that this jump signature is purely an obfuscation and that the first legitimate call is the local address towards the beginning of the signature.

4.2.3. Limitations of EtherAnnotate Analysis. Although the EtherAnnotate provides valuable, new information about what string values registers hold during runtime, there are some situations that the plugin currently does not handle. The situations where EtherAnnotate currently cannot resolve the data stored in a memory location can be separated into two instruction operand categories: complex register referencing and direct memory access. Complex register referencing encompasses all instructions with operands where the registers are not directly referenced (e.g., `MOV [EAX+EBP], 0x05` would be a complex register reference). This analysis deficit is purely implementation-specific and can be resolved by adding extra functionality to the Xen-side of the EtherAnnotate plug-in that will recognize more libdisasm disassembly operands. Similarly, direct memory access encompasses all instructions with operands that refer to memory addresses instead of processor registers (e.g., `PUSH DS:[0x432C03]`). These instructions may or may not have additional operands that are registers, but they all directly access a memory location without going through a register. Like the complex register referencing, additional functionality would need to be added to handle the libdisasm parsing for direct memory access. Additionally, since the current plug-in only pulls register data from Xen, more functionality would have to be added to map the Xen virtual machine's virtual memory into the actual host memory. This would allow the EtherAnnotate plug-in to directly access what value is stored at that address.

5. CASE STUDY OF SPYEYE CYBERCRIME TROJAN

This section presents a second case study, focusing on how analysis using EtherAnnotate compares to current dynamic and static analysis tools. The target of this case study is a binary created by the cybercrime software toolkit (crimeware) called SpyEye. The SpyEye toolkit is sold for \$500 on Russian underground forums and allows an attacker to build a custom Trojan bot that can capture a wide array of personal information once installed on a victim's computer and send the information back to a central server [2]. Section 5.1 describes the process used to decide to evaluate EtherAnnotate on the SpyEye Trojan and why the other candidate malware were not able to be analyzed. Section 5.2 summarizes the previous reverse engineering work done on SpyEye by security labs and researchers and Section 5.3 presents the new analysis using EtherAnnotate. Finally, Section 5.4 gives a study of EtherAnnotate's benefits compared to the other commonly used malware analysis tools.

5.1. FINDING A MALICIOUS BINARY FOR COMPARATIVE ANALYSIS

The goal of this case study was to show the benefits and differences of the EtherAnnotate analysis tool compared to other commonly used static and dynamic analysis tools. The majority of reverse engineering information on malicious software is located on security labs' websites and blogs as well as on a few various Internet forums focused on malware analysis and reverse engineering. One especially useful site for finding malware to study is an online malicious binary repository hosted on Offensive Computing¹⁴ as well as a forum for discussing malware and requesting samples. On this site, a security researcher can search by a file's hash, filename, or anti-virus signature name to find over a million different malicious binaries stored on their servers. However, finding a malicious file that executes easily (e.g., does not need a server to be up, does not inject into other processes, etc...) and behaves in ways to hinder analysis is not a simple challenge. The search for a binary for the second

¹⁴<http://offensivecomputing.net/>

case study was narrowed down to a botnet by the name Conficker and a crimeware Trojan named SpyEye, which their variants are described as:

- **Conficker.A:** The first version of the Conficker worm, runs as an EXE file but did not perform noticeable malicious activity.
- **Conficker.C:** A later version of Conficker that uses P2P communication, does not allow dynamic analysis tools to run and is a DLL that needs to be started as a service.
- **SpyEye:** - A new crimeware Trojan EXE that spawns a remote thread in explorer.exe and then communicates back to a command & control server.

The Conficker.A worm, released in early 2009, was the first that the author looked at as a candidate because of the large amount of previous research done on Conficker and its variants [34]. However, after running the sample¹⁵ in both a VMware sandbox and inside of Xen EtherAnnotate, neither environment showed evidence that the worm was behaving maliciously. This was deduced by looking at Wireshark network traces, Process Monitor output, RegShot output, and the EtherAnnotate trace file. Additionally, even after the Conficker.A variant had been unpacked from its original UPX packing, the EtherAnnotate trace file was over 10GB which was deemed to be too large for practical analysis in the IDA Pro plug-in (this was before the plug-in was ported to C++ as described in Section 3.4.2).

The next candidate malware that was examined was the Conficker.C variant, also released in 2009, since there was a relatively large amount of previous research on its behavior as well [35]. The Conficker.C variant was delivered as a DLL library¹⁶ instead of an executable file and was designed to be dropped into the `C:\Windows\System32` directory and loaded as a service. The Ether instruction tracer was not designed specifically for tracing DLL files since their execution trace comes from another program loading the DLL into memory and calling its functions, but attempts were made by the author to allow for DLL tracing. The Conficker.C variant was able to be loaded in the Xen environment by moving the

¹⁵MD5: d60960adb601613ec330eb36690ea59e

¹⁶MD5: 5e279ef7fcb58f841199e0ff55cdea8b

DLL file into the System32 directory, changing a non-essential service's (Network Location Awareness) configuration in the Windows Registry to point to the new Conficker DLL, and rebooting the virtual machine. Upon restart, the worm would attempt to contact command & control servers as seen in Wireshark network trace logs and would cause dynamic analysis programs such as Process Monitor and Process Explorer to automatically close upon execution. However, the Conficker.C's execution was unable to be traced in this method since Windows uses the `svchost.exe` executable to load multiple Windows services DLLs - tracing `svchost.exe` would produce a trace containing interleaved execution of all services running under that process. Two workarounds were designed that allowed a single service to be running under an instance of the `svchost.exe` binary; however, the Conficker.C DLL would load under neither of these workarounds. The first workaround consists of changing the value in the Windows registry for the non-essential service such that it loads the DLL with a renamed copy of `svchost.exe` instead of the original. By changing the value of `\HKLM\SYSTEM\CurrentControlSet\Services\<Service Name>\ImagePath` to point to the renamed copy of `svchost.exe` instead of the original path. This modification will cause the service to load the Conficker.C DLL at startup and will be loaded by its own, personal instance of `svchost.exe`. The second workaround consists of creating a new service group that only contains the Conficker.C DLL service which causes one of the instances of `svchost.exe` to be running only the Conficker.C code [36]. Both of these workarounds require the *TimeoutPeriod* value to be added to the registry as described in [37] so that the service does not timeout while the process is being single-stepped.

These changes did allow for benign services to be loaded by themselves and traced, but the Conficker.C refused to be loaded when any of these changes were made to the system. Further research would need to be done in order to determine the best methods for tracing DLL services in Windows using the Xen Ether tracer without modifying the underlying guest system.

The final malware candidate was the SpyEye Trojan which is a custom-built binary file that can be generated by purchasing the SpyEye building software from Russian sources. This file is an EXE executable and was able to run and cause malicious behavior while under analysis although it only attempted to contact the

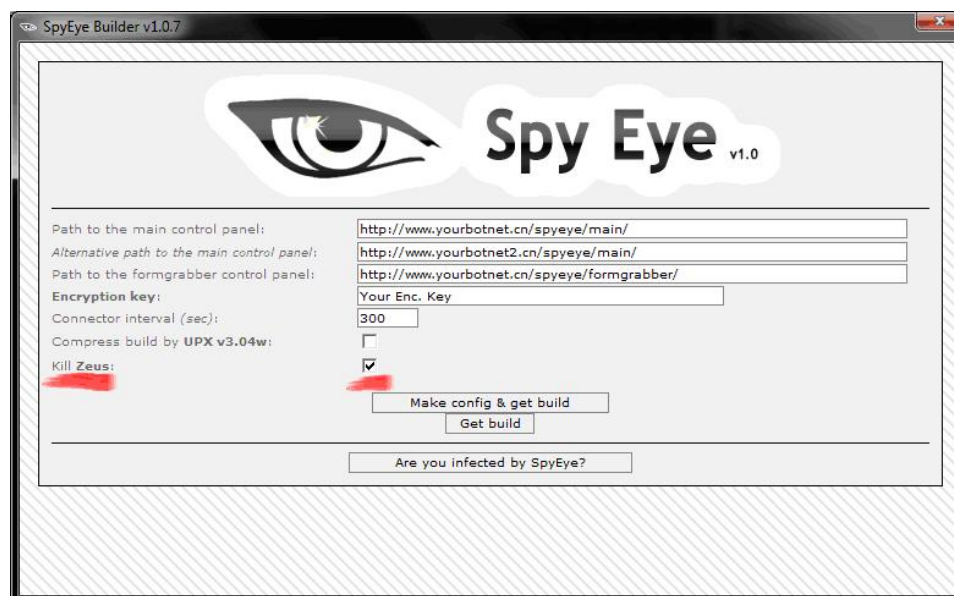


Figure 5.1. A screenshot of the SpyEye Trojan builder [2]

command & control servers, which have been shut down, and then went dormant. Additionally, although this Trojan has only been active since the beginning of 2010 there is a relatively large number of reverse engineering reports that focus mostly on the builder but also on some of its communication [2][38][39]. Figure 5.1 provides a sample screenshot of the SpyEye builder that an attacker would use to generate the SpyEye Trojan that will be examined in this section. This thesis presents a new, in-depth look into the actions of the SpyEye Trojan as it loads itself into memory and checks for previous infections.

5.2. PREVIOUS ANALYSIS OF SPYEYE TROJAN

5.2.1. Trojan Installation. The Trojan is typically installed in the wild using the TrojanDropper:Win32/Spyeye malicious binary that a user has been convinced to execute on their system. This dropper binary connects to a server and downloads the Trojan:Win32/Spyeye binary that will be analyzed in this section [40]. According to the Microsoft Malware Protection Center's report on SpyEye, the malware installs itself in the `%SystemDrive%\cleansweep.exe\cleansweep.exe` path

and creates a mutex named “_CLEANSWEEP_” to guarantee that only one instance of the Trojan is running at a time. Additionally, the Trojan creates a value in the Windows registry that instructs the operating system to load cleansweep.exe on system start-up and also injects itself into currently running processes.

5.2.2. Trojan Payload. Once the SpyEye malware has installed itself on the system and injected itself into one of the processes currently running, it starts to take actions to hide itself and steal sensitive user data. The Trojan manages to hide the actual process by injecting itself into another process and executing there; the Trojan hides its files and registry values by using rootkit techniques to hook low-level APIs and modify the query results for files and registry values. The SpyEye binary uses similar techniques to hook various system functions that handle HTTP traffic and records login information such as form data and keystrokes. Finally, once the Trojan has captured data, it will send the data to various command & control servers (most likely the servers hosted by whoever built the Trojan using the SpyEye toolkit).

5.3. ETHERANNOTATE ANALYSIS OF SPYEYE TROJAN

The SpyEye Trojan binary that was analyzed using the EtherAnnotate toolset was downloaded from the Offensive Computing repository¹⁷ and was packed with the UPX packing software. Once this case study began, the author realized that the current implementation of EtherAnnotate was too slow to finish in a practical amount of time and key changes were made to both the Xen and IDA Pro aspects of the toolset and are described in Section 3.4. The malware was initially unpacked using QUnpack in order to reduce the size of the EtherAnnotate trace file and speed up the IDA Pro annotations. However, after running the malware and looking at the annotated disassembly in IDA Pro it became apparent that the malware is able to detect that it had been modified and unpacked. Figure 5.2 shows the area in code where this check occurs and it is apparent from the code coverage visualization that instead of calling `CreateRemoteThread()` the code instead jumps past the call. The call to `CheckFileModifications` shown in Figure 5.2 was renamed by the author during analysis to help identify key areas of the disassembly and the internals of the function consist of multiple calls to `VirtualQueryEx()`. `VirtualQueryEx()` is a Windows API

¹⁷MD5: 9d2a48be1a553984a4fda1a88ed4f8ee

```

push [ebp+hProcess] ;
call CheckFileModifications ;
mov esi, eax ; esi: 0x7c80b929 eax: 0xffffffff
cmp esi, ebx ; esi: 0xffffffff ebx: 0x00000000
pop ecx ; ecx: 0x00000001
jl short loc_403997 ;

loc_403977:
cmp [ebp+arg_8], ebx
jz short loc_403977

mov eax, [ebp+arg_C]
mov [ebp+var_2C], eax
mov [ebp+var_3C], esp
mov eax, [ebp+var_2C]
push eax
call [ebp+arg_8]
mov esp, [ebp+var_3C]

loc_403977:
lea eax, [ebp+ThreadId]
push eax ; lpThreadId
mov eax, [ebp+arg_4]
push ebx ; dwCreationFlags
push esi ; lpParameter
add esi, eax
push esi ; lpStartAddress
push ebx ; dwStackSize
push ebx ; lpThreadAttributes
push [ebp+hProcess] ; hProcess
call ds:CreateRemoteThread

```

Figure 5.2. A screenshot of the SpyEye check for file modifications

function that retrieves information about pages within the virtual address space of the specified process - in this case, it is examining its own pages for modifications. Upon noticing this anti-analysis technique, the SpyEye Trojan was run through the Xen EtherAnnotate tool without unpacking it first. The trace was loaded into IDA Pro again and this time the code coverage visualization showed that the binary did not jump around the `CreateRemoteThread()` call and instead made the call using values from `explorer.exe`. Additionally, HTTP queries were noticed in the Wireshark logs to the servers `textttvinodelam.nett.cn` and `fw.ename.cn` which match the description of previous research [39].

5.3.1. Initialization and Anti-Tamper Checks. Once a trace was created that followed the SpyEye malware through its malicious behavior, the trace was loaded into IDA Pro using the EtherAnnotate C++ plug-in and its behavior was analyzed. The malware first checks if there is already a system mutex with the name “_CLEANSWEEP_”, as described in previous research, and jumps around the malicious code if there is - system mutexes are commonly used by malware to leave their fingerprint on infected machines. The core infection routine happens at 0x403754 and consists of modification of the file as described above, checks to make sure the code is going to be injected into certain system files, and the `CreateRemoteThread()` call that performs the injection. The code first grabs the name of the process in which the code is executing and then goes through a series of calls to the string comparison function `lstrcmpiA()` to check that the code is not going to be injected into certain system files. Figure 5.3 shows a screenshot from IDA Pro with the EtherAnnotate annotations during part of the system file check sequence.

5.3.2. Injection into Explorer.exe. The next key aspect of the initialization sequence is to find the base memory address of the process that SpyEye wants to inject itself into. The code for this begins at 0x403469 and consists of a call to `VirtualQueryEx` to retrieve the base address and then a loop which calls `WriteProcessMemory` that copies the SpyEye payload into explorer.exe’s memory. If the analyst takes notice to the values returned by `VirtualQueryEx` and the arguments passed into `WriteProcessMemory` in Figure 5.4, it becomes apparent that the base address for explorer.exe is 0x0e600000. This is important to notice since it confirms that `WriteProcessMemory` is writing data from the SpyEye binary’s own memory into a location in explorer.exe’s memory, but also confirms later that the call to `CreateRemoteThread` passes in a function from explorer.exe as the `lpStartAddress` argument.

Figure 5.5 shows the call to `CreateRemoteThread` with the EtherAnnotate register value annotations as well as the function argument names added by the author. From the previous realization that 0x0e600000 is the base address for explorer.exe and that SpyEye wrote portions of its code to this process, it can be determined that the `lpStartAddress` argument of 0x0ea6148a is the location of that copied code. Browsing in IDA Pro to 0x0040148a in the SpyEye disassembly will reveal the instructions that were copied into explorer.exe and executed under that process. However, since


```

push  0CEE1148Dh ;
call  sub_401000 ;
mov   esi, ds:!strcmpi0 ; esi: 0x7c80b929
pop   ecx ; ecx: 0x00003ad5
push  eax ; eax: 0x00405478 ptr_val[]: System
lea   eax, [ebp+String1] ; eax: 0x00405478 ptr_val[]: System,
push  eax ; eax: 0x0012fc94 ptr_val[]: wuauclt.exe
call  esi ; !strcmpi0 ; esi: 0x7c80b929
test  eax, eax ; eax: 0x00000001 eax: 0x00000001
jz    loc_403997 ;

push  4A9736A0h ;
call  sub_401000 ;
ecx   ecx ; ecx: 0x00003ad5
push  eax ; eax: 0x004054dc ptr_val[]: smss.exe
lea   eax, [ebp+String1] ; eax: 0x004054dc ptr_val[]: smss.exe,
push  eax ; eax: 0x0012fc94 ptr_val[]: wuauclt.exe
call  esi ; !strcmpi0 ; esi: 0x7c80b929
test  eax, eax ; eax: 0x00000001 eax: 0x00000001
jz    loc_403997 ;

push  3DC4AC21h ;
call  sub_401000 ;
ecx   ecx ; ecx: 0x00003ad5
push  eax ; eax: 0x00405540 ptr_val[]: csrss.exe
lea   eax, [ebp+String1] ; eax: 0x00405540 ptr_val[]: csrss.exe,
push  eax ; eax: 0x0012fc94 ptr_val[]: wuauclt.exe
call  esi ; !strcmpi0 ; esi: 0x7c80b929
test  eax, eax ; eax: 0x00000001 eax: 0x00000001
jz    loc_403997 ;

```

Figure 5.3. A screenshot of the SpyEye injection checks

the execution was carried out under the explorer.exe process and not the SpyEye.exe process, EtherAnnotate is currently unable to trace those instructions since it does process tracing by a filename and CR3 filter.

5.4. COMPARISON OF ANALYSIS TOOLS

The SpyEye Trojan was additionally analyzed using a variety of common analysis tools and techniques to evaluate how EtherAnnotate compares in versatility. The malware was loaded onto a Windows XP SP2 virtual machine along with an Ubuntu Linux virtual machine in a VMware team. Running inside of a virtual machine was required for these experiments in order to make containment of malicious behavior easier and more reliable than running on bare-metal. The Linux virtual machine was running Wireshark to monitor network traffic and was configured to act as the router

```

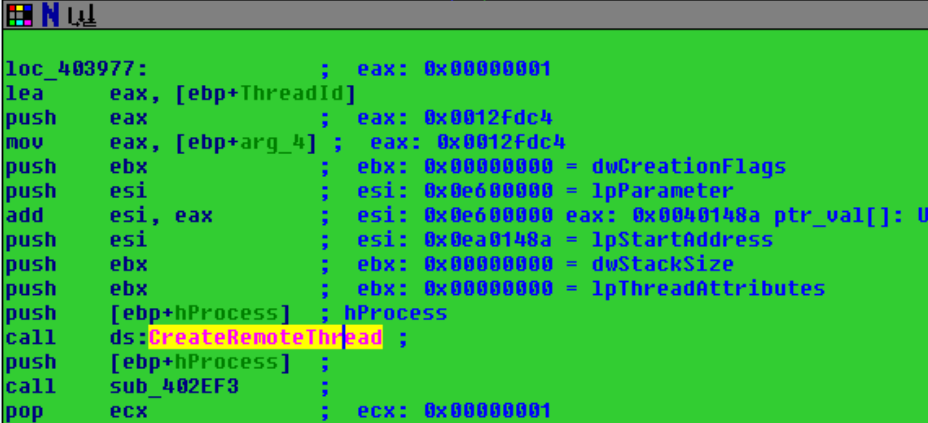
and     eax, 0FFFh      ; eax: 0x0000331c
add     eax, [esi]      ; eax: 0x0000031c
add     eax, [ebp+var_4] ; eax: 0x0002831c
mov     ebx, eax        ; ebx: 0x0ea28318  eax: 0x0ea2831c
lea     eax, [ebp+NumberOfBytesWritten] ; eax: 0x0ea2831c
push    eax             ; eax: 0x0012fa34
push    4               ;
lea     eax, [ebp+lpAddress] ; eax: 0x0012fa34
push    eax             ; eax: 0x0012fa44 ptr_val[]: 9
push    ebx             ; ebx: 0x0ea2831c
push    [ebp+hProcess] ;
call    ds:ReadProcessMemory ;
mov     edi, eax        ; edi: 0x00000001  eax: 0x00000001
mov     eax, [ebp+var_18] ; eax: 0x00000001
add     [ebp+lpAddress], eax ; eax: 0x0e600000
lea     eax, [ebp+NumberOfBytesWritten] ; eax: 0x0e600000
push    eax             ; eax: 0x0012fa34
push    4               ;
lea     eax, [ebp+lpAddress] ; eax: 0x0012fa34
push    eax             ; eax: 0x0012fa44
push    ebx             ; ebx: 0x0ea2831c
push    [ebp+hProcess] ;
and     edi, 1         ; edi: 0x00000001
call    ds:WriteProcessMemory ;
test    edi, eax        ; edi: 0x00000001  eax: 0x00000001
jz     short loc_403625 ;

```

Figure 5.4. A screenshot of SpyEye injecting its code into explorer.exe

for the Windows virtual machine. Additionally, the Linux guest was configured to respond to all DNS requests with its own IP address instead of querying an actual DNS server for the network. This DNS spoofing allows for network aliveness checks in the Windows guest to at least succeed, making it appear that the host the guest program is looking for is alive but not doing anything.

The first analysis experiment was to run the SpyEye Trojan on the Windows guest with no analysis programs running. This was done to achieve a control group (at least for network behavior) in case other analysis tools caused the SpyEye malware to change its behavior. The Wireshark network trace was similar to that captured during EtherAnnotate analysis; five identical POST requests and one GET request are made to the vinodelam.nett.cn domain. The POST requests contain a sequence of formatted messages containing bot identifiers, time zone information about the guest, language information about the guest, and the Windows OS version of the guest. The GET request appears to be a registration scheme for the command & control server; it sends the ID of the guest, version information about programs on the guest, and



```

loc_403977:          ; eax: 0x00000001
lea     eax, [ebp+ThreadId]
push   eax           ; eax: 0x0012f4c4
mov    eax, [ebp+arg_4] ; eax: 0x0012f4c4
push   ebx           ; ebx: 0x00000000 = dwCreationFlags
push   esi           ; esi: 0x0e600000 = lpParameter
add    esi, eax      ; esi: 0x0e600000 eax: 0x0040148a ptr_val[]: 0
push   esi           ; esi: 0x0ea0148a = lpStartAddress
push   ebx           ; ebx: 0x00000000 = dwStackSize
push   ebx           ; ebx: 0x00000000 = lpThreadAttributes
push   [ebp+hProcess] ; hProcess
call   ds:CreateRemoteThread ;
push   [ebp+hProcess] ;
call   sub_402EF3    ;
pop    ecx           ; ecx: 0x00000001

```

Figure 5.5. A screenshot of SpyEye creating a new thread inside explorer.exe

a CRC identifier for the bot. These protocol exchanges were previously published in the Microsoft Malware Protection Center report [40].

The second experiment was to run the SpyEye Trojan inside of the stock OllyDbg debugger to determine if any anti-debugging mechanisms are present in SpyEye. The Linux guest was running Wireshark as before and recorded another similar network trace. From this experiment, it was deduced that the SpyEye binary does not contain any obvious debugger detection mechanisms; although, the program was executed without any breakpoints or single-stepping which can cause certain detections to occur. For the third experiment, the SpyEye binary was loaded on the Windows guest again and this time monitored using Process Monitor and Regshot, providing information on event logging and registry changes respectively. The Process Monitor results showed that SpyEye wrote two files, `C:\cleansweep.exe\config.bin` and `C:\cleansweep.exe\cleansweep.exe`, and then suddenly explorer.exe process executes the cleansweep.exe file. From the previous analysis with EtherAnnotate, this event becomes clear that explorer.exe is performing malicious activity because SpyEye did a code injection attack on the process. The files written to disk were referenced in previously published analysis [40]. Regshot did not provide any registry changes other than those typical of launching a program in Windows XP. Finally, the

10.10.10.1	DNS	Standard query A vinodelam.nett.cn
10.10.10.69	DNS	Standard query response A 10.10.10.1
10.10.10.1	TCP	icp > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460
10.10.10.69	TCP	http > icp [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1460
10.10.10.1	TCP	icp > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
10.10.10.1	HTTP	POST http://vinodelam.net/spy/formgrab/websitechk.php HTTP/1.1
10.10.10.69	TCP	http > icp [ACK] Seq=1 Ack=857 Win=6848 Len=0
10.10.10.1	DNS	Standard query A vinodelam.nett.cn
10.10.10.69	HTTP	HTTP/1.1 404 Not Found (text/html)
10.10.10.69	TCP	http > icp [FIN, ACK] Seq=499 Ack=857 Win=6848 Len=0

Figure 5.6. Snapshot of Wireshark Network Trace with Control Experiment

SpyEye binary was run through PEiD in order to see what signature identification could be produced for the packer and PEiD reported that UPX 3.0 had been used.

Although there appear to be no anti-analysis techniques employed in the SpyEye binary, the information that EtherAnnotate provides goes beyond the basic analysis possible with previous tools. The EtherAnnotate trace could not follow beyond the code injection into explorer.exe, but the annotations from before the injection provided detailed information on how the malware behaved. The check for file consistency was not apparent in the information provided by the previous tools and was useful to determine the conditions in which SpyEye executes. Additionally, the process for determining which process to inject, finding the base memory address of that process, and injecting the code into memory allocated inside of the guest process are complex procedures that EtherAnnotate described and could potentially be used to identify future malware.

6. CONCLUSIONS

6.1. PRESENTED SOLUTION TO MALWARE ANALYSIS PROBLEM

This thesis has described EtherAnnotate, a research tool for malicious software security analysts which presents a merger of information gathered in dynamic analysis and visualized in a static analysis disassembly view. By applying the EtherAnnotate tool to a malicious binary program, a researcher is able to statically view what values variables and memory addresses the program held during its actual runtime. Furthermore, since EtherAnnotate's tracing module is built upon the Xen Ether framework, it is transparent to the execution of the malware for all anti-debugging attacks and most anti-VM attacks. The insight that will be obtained by using this novel toolset will hopefully allow future malware analysts to better detect and defend against the increasingly advanced malicious attacks on computer systems.

6.2. STRENGTHS OF THE CURRENT SYSTEM

EtherAnnotate is unique in that it allows dynamic malware analysis that is completely undetectable by the common anti-debugging malware obfuscations. Because the Ether framework has a higher privilege than both the guest kernel and guest applications, EtherAnnotate can monitor any actions of malicious binaries without the need to modify the state of the guest operating system. The only possible known attacks against Ether and thus EtherAnnotate's transparency are by detecting virtualized devices and detecting timing inconsistencies. The properties and behavior of the host system's devices can be passed through to the guest or potentially emulated by the hypervisor with additional implementation work. Local timing attacks for virtualization detection can be subverted by utilizing the Intel VT-x TSC offset which causes the guest's virtual processor to return purposefully skewed clock time in order to hide the additional time taken during VM-exits. Xen currently utilizes the VT-x TSC offset feature, but checking the time using remote data is impossible to easily detect or defend against. Since EtherAnnotate is transparent to the guest

operating system in these ways, its presence as a tool in the malware analysts repertoire will force malware authors to fall back to anti-VM detection techniques that rely on remote timing analysis in order to evade detection. However, using anti-VM techniques is becoming less and less appealing to malware authors as more companies and individuals start using virtual machines. A larger virtualization user-base eliminates a substantial number of possible adversarial targets since VM-detection only detects that a VMM is present, not that analysis is occurring. This compromise and restriction of available obfuscation actions of malicious binaries is one of the more interesting and open areas of research in the virtualization introspection field.

EtherAnnotate's integration of data collected during dynamic analysis with the static disassembly generated by IDA Pro opens new possibilities for in-depth analysis. Previously, the most common method to visually inspect the run-time values of a program with the disassembled instructions would be to run a trace of the program using an application such as OllyDbg and stepping through each line of code to see the values at each line. The tools presented in this thesis improve on previous integration techniques in three ways: transparent tracing, visual trace annotations, and heuristics for data recording. The transparent tracing benefits described in the previous section provide the malware analyst with a guarantee that the malware under analysis will have identical behavior inside of EtherAnnotate as it would on a bare-metal environment (other than possible anti-VM detection). There are tools, such as plug-ins for OllyDbg, which can guard against anti-debugging detection but these previous tools can only defend against known techniques and are not guaranteed to work in all cases - EtherAnnotate gives the analyst this guarantee. The closest visual tracing tool currently available is probably IDA Pro's recently added tracing ability shown in Figure 6.1. The IDA Pro tracing functionality provides a record of all instructions executed and a list of register values for all operands of these instructions. It does not follow register values to detect possible pointers to strings and the IDA Pro debuggers do not have any defenses against anti-debugging attacks. For example, the trace shown in Figure 6.1 was from a benign program which utilized many different anti-debugging techniques and crashed before unpacking itself by detecting the breakpoints IDA Pro had set. Additionally, the IDA Pro trace appears in a separate window from the actual disassembly and the end-user analyst must toggle

```

0000021C .rsrc:004459DE          push     edx                ESP=1054FF70
0000021C .rsrc:004459DF          mov     esi, eax           ESI=20E78
0000021C .rsrc:004459E1          mov     eax, [esi-4]      EAX=1A0
0000021C .rsrc:004459E4          add     eax, 4            EAX=1A4 PF=0
0000021C .rsrc:004459E7          sub     esi, eax          ESI=20CD4 PF=1
0000021C .rsrc:004459E9          mov     [esi+8], edx
0000021C .rsrc:004459EC          mov     ecx, [ebx+0Ch]    ECX=45883
0000021C .rsrc:004459EF          mov     [esi+14h], ecx
0000021C .rsrc:004459F2          call    edi               call 00020E78
0000021C .rsrc:004459F2          call    edi               EAX=43E000 ECX=3D2C0000 EDX=8E3B8
0000021C .rsrc:004459F4          mov     [ebp+1000133Fh], eax
0000021C .rsrc:004459FA          mov     esi, eax           ESI=43E000
0000021C .rsrc:004459FC          mov     ecx, [ebx+14h]    ECX=4493C
0000021C .rsrc:004459FF          pop     edx               EDX=400000 ESP=1054FF74
0000021C .rsrc:00445A00          jmp     short loc_445A0E
0000021C .rsrc:loc_445A0E          mov     eax, esi
0000021C .rsrc:00445A10          pop     edx               EDX=414790 ESP=1054FF78
0000021C .rsrc:00445A11          pop     esi               ESI=0 ESP=1054FF7C
0000021C .rsrc:00445A12          pop     edi               EDI=0 ESP=1054FF80
0000021C .rsrc:00445A13          pop     ecx               ECX=0 ESP=1054FF84
0000021C .rsrc:00445A14          pop     ebx               EBX=7EFDE000 ESP=1054FF88
0000021C .rsrc:00445A15          pop     ebp               EBP=1054FF94 ESP=1054FF8C
0000021C .rsrc:00445A16          jmp     eax

```

Figure 6.1. Example Instruction Trace from IDA Pro’s Debugger

between the two windows to correlate what the register values were at each instruction - having this information on a single window using EtherAnnotate’s comments seems much more practical. Finally, the last major strength of EtherAnnotate is the heuristic for determining run-time string values and visualizing them in the IDA Pro disassembly. If a botnet binary is going to decrypt and encrypt data that it receives and sends to its Command & Control server, EtherAnnotate can capture the plaintext strings before being encrypted. A malware researcher analyzing the code can follow the program’s flow using the colorized code coverage and use the annotations provided to understand the botnet’s protocol and walk through how the encryption algorithm works.

6.3. ISSUES THAT LIMIT USEFULNESS

Although EtherAnnotate provides many new opportunities to combine runtime variable values with a graphical disassembly for better understanding of malware, there are many areas of the toolset that can be improved for better usability and better analysis. The first area for improvement is the overall speed of analysis - the bug fix described in Section 3.4.1 and the port to C++ described in Section 3.4.2 tremendously increased performance, but more can be done. After running many

EtherAnnotate traces on different types of files, the author noticed that although Ether starts to single-step immediately after a program is executed, the instruction trace does not record anything for approximately the first minute. During this time, it appears that the Windows executable loader code is allocating memory and initializing data structures for the binary that is about to execute. Further research should be done in determining the process that Windows uses to load an executable before the code flow is handed off to the executable's entry point - if this loading process is what's taking up time before a trace, it should not be single-stepped. In addition, although some runtime performance analysis was recorded in Section 3.2.2, it would be interesting to record more data more accurately in order to provide benchmarks for future research.

One of the main limitations of EtherAnnotate for analysis purposes is the lack of support for certain classes of instructions. Currently, the only instruction classes that are supported for analysis are instructions that directly reference register names (e.g., `MOV EAX, EBX` or `PUSH ECX`). This is due to the prototype implementation that EtherAnnotate originated from and currently only does simple string parsing to determine what registers are present in the instruction mnemonic. It should be fairly easy to add support for all instruction classes, such as registers with mathematical operations (e.g., `MOV [EAX+0x4], ESP`), in the future by creating more robust parsing algorithms for the *libdisasm* mnemonics provided in Xen Ether. Additionally, the current code only searches for values for CPU registers but many instructions directly reference memory locations. However, it turns out that this is not a true limitation since the x86 instruction set does not allow direct manipulation of memory values. Memory values must first be loaded into a general purpose register, manipulated using this register, and then the result is stored back into a memory address. Because of this process, only recording the register values will in fact capture all of the pertinent data of the operations.

The limitation that was most obvious during this thesis's case studies was that the instruction tracing ability only works for a single process's execution. This is not a problem for most applications, but malware will often inject its code into a second process and hide its continuing execution there. The SpyEye Trojan did exactly this and hid its truly malicious activity inside of `explorer.exe`. A start to finding a usable

solution to this problem would be to monitor system calls made by the process and record the information passed to `CreateRemoteThread` class (or any other calls that start code flow in a separate process). Once a call is made that spawns a thread or process in a separate binary file, the injected process's name can be recorded and a separate `EtherAnnotate` trace could start on this process. Additionally, a memory dump of the injected process would need to be made in order to obtain the executable with the injected code intact. These additions to `EtherAnnotate` should not prove to be too difficult since `Ether` provides methods for system call tracing and its unpacking tool makes memory dumps of processes at runtime.

6.4. FUTURE WORK

`EtherAnnotate` is one of the first extensions to the Xen `Ether` framework which will hopefully expand to provide more analysis tools in new areas that were not previously possible. One of the areas which has the most potential for future research is designing smarter heuristics for data collection during the Xen `EtherAnnotate` trace algorithm. Currently, the algorithm simply checks if a register's value points to a character string array of ASCII printable bytes up to a certain length. During design, research was started to determine what signature the memory allocation and memory free Windows system calls leave on the memory they affect. Identifying the structures left in around these memory allocations could help provide a more accurate recognition of string values. However, compilers that do not follow the Microsoft standards for string allocation, or programs written in raw assembly, will not have these same signatures so a generic heuristic is also required. Additionally, many Windows API functions will accept and return Unicode strings instead of ASCII strings. The Unicode character set is 16-bits wide compared to 8-bit ASCII and thus contain a much larger set of possible characters. When the printable ASCII characters are represented in Unicode, each letter's lower 8-bits are equivalent to the ASCII-encoded character and the upper 8-bits are `0x00`. These strings will currently not be recorded by the string parsing heuristic since it stops recording a string when it detects a NULL byte (`0x00`).

Now that there is a transparent system for recording runtime values of malicious binaries, future research into what can be correlated from this data may prove useful in

classification. Most current malware classification systems work by building statistical correlations of the data present inside of a malicious binary file to other previously seen binary files. Another classification method is to attempt the same type of correlation based on the network activity or other observable events caused by a binary. Both of these areas have shown promise in previous research and would be cause to examine the potential of using internal runtime data for program correlation and attribution. For instance, a malicious botnet binary may send and receive encrypted traffic to a centralized command & control server that would not easily correlate to other instances on the botnet since each message appears random. However, by monitoring the application in EtherAnnotate the data should appear as a string variable at the point in the binary where encrypted network traffic is decrypted. This additional information could prove useful for increasing the resilience of malware identification and attribution.

BIBLIOGRAPHY

- [1] D. Quist and L. Liebrock, “Visualizing compiled executables for malware analysis,” in *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, Oct. 2009, pp. 27–32.
- [2] M. B. Barcena, “SpyEye Bot versus Zeus Bot,” Symantec Corporation, Tech. Rep., Feb. 2010. [Online]. Available: <http://www.symantec.com/connect/blogs/spyeye-bot-versus-zeus-bot>
- [3] J. Walker, “The animal episode: Pervade source code,” Fourmilab, Tech. Rep., 1996. [Online]. Available: <http://www.fourmilab.ch/documents/univac/pervade.html>
- [4] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.
- [5] Z. Chen, L. Gao, and K. Kwiat, “Modeling the spread of active worms,” 2003, pp. 1890–1900.
- [6] S. Corporation, “Internet security threat report,” Symantec Corporation, Tech. Rep., 2009.
- [7] A. Press, ““zombie” ring allegedly hit 1.5 million computers,” *MSNBC*, Oct. 2005. [Online]. Available: <http://www.msnbc.msn.com/id/9763824/>
- [8] P. Ferrie, “Anti-unpacker tricks,” in *2nd International CARO Workshop*, May 2008. [Online]. Available: <http://pferrie.tripod.com/papers/unpackers.pdf>
- [9] —, “Attacks on more virtual machine emulators,” Symantec Advanced Threat Research, Tech. Rep., 2007. [Online]. Available: <http://pferrie.tripod.com/papers/attacks2.pdf>
- [10] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not transparency: Vmm detection myths and realities,” in *HOTOS’07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–6.

- [11] D. H. Barnes and L. L. Wear, "Instruction tracing via microprogramming," in *MICRO 7: Conference record of the 7th annual workshop on Microprogramming*. New York, NY, USA: ACM, 1974, pp. 25–27.
- [12] C. C. Zou, L. Gao, W. Gong, and D. Towsley, "Monitoring and early warning for internet worms," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2003, pp. 190–199.
- [13] U. Bayer, "TTAnalyze: A Tool for Analyzing Malware," Master's thesis, Technical University of Vienna, Dec. 2005. [Online]. Available: http://www.iseclab.org/people/ulli/TTAnalyze_A_Tool_for_Analyzing_Malware.pdf
- [14] S. Fox, "Cyber warfare and attribution," *CSO Security and Risk*, Jul. 2009. [Online]. Available: http://blogs.csoonline.com/cyber_warfare_and_attribution
- [15] D. A. Wheeler and G. N. Larsen, "Techniques for cyber attack attribution," Institute for Defense Analysis, Tech. Rep., Oct. 2003.
- [16] R. J. Turk, "Cyber incidents involving control systems," US-CERT Control Systems Security Center, Tech. Rep., Oct. 2005.
- [17] *Tor: The Second-Generation Onion Router*. USENIX Security Symposium, 2004.
- [18] T. Abou-assaleh, N. Cercone, and R. Sweidan, "R.: Detection of new malicious code using n-grams signatures," in *In: Proc. Second Annual Conference on Privacy, Security and Trust*, 2004, pp. 13–15.
- [19] L. H. Seawright and R. A. MacKinnon, "Vm/370 - a study of multiplicity and usefulness," International Business Machines, Tech. Rep., 1979.
- [20] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," International Business Machines, Tech. Rep., 1981.

- [21] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.
- [22] R. P. Goldberg, “Architectural principles for virtual computer systems,” Harvard University - Division of Engineering and Applied Physics, Tech. Rep., 1973.
- [23] J. N. Matthews, E. M. Dow, T. Deshane, W. Hu, J. Bongio, P. F. Wilbur, and B. Johnson, *Running Xen: A Hands-On Guide to the Art of Virtualization*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
- [24] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.
- [25] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3B*, Intel Corporation, Sep. 2008.
- [26] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3A*, Intel Corporation, Sep. 2008.
- [27] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, “Ether: malware analysis via hardware virtualization extensions,” in *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2008, pp. 51–62.
- [28] VMware, Inc., “VMware Virtualization Software for Desktops, Servers & Virtual Machines for a Private Cloud,” 2010. [Online]. Available: <http://www.vmware.com/>
- [29] Oracle Corporation, “VirtualBox,” 2010. [Online]. Available: <http://www.virtualbox.com/>
- [30] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, “Automatic Reverse Engineering of Malware Emulators,” May 2009, pp. 94 –109.
- [31] Oreans Technologies, “Themida: Advanced Windows Software Protection System,” 2009. [Online]. Available: <http://www.oreans.com/themida.php>

- [32] VMProtect Software, “VMProtect - New-generation software protection,” 2010. [Online]. Available: <http://www.vmprotect.ru/>
- [33] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 1*, Intel Corporation, Sep. 2008.
- [34] P. Porras, H. Saidi, and V. Yegneswaran, “An Analysis of Conficker’s Logic and Rendezvous Points,” SRI International, Tech. Rep., Mar. 2009. [Online]. Available: <http://mtc.sri.com/Conficker/>
- [35] —, “Conficker C Analysis,” SRI International, Tech. Rep., Apr. 2009. [Online]. Available: <http://mtc.sri.com/Conficker/>
- [36] MSDN, “Preparing to Debug the Service Application,” Windows Developer Center, Tech. Rep., Mar. 2010. [Online]. Available: <http://msdn.microsoft.com/en-us/library/cc835863.aspx>
- [37] —, “How to debug Windows services,” Windows Developer Center, Tech. Rep., Mar. 2010. [Online]. Available: <http://support.microsoft.com/kb/824344>
- [38] J. Mieres, “SpyEye Bot. Analysis of a new alternative scenario crime-ware,” Feb. 2010. [Online]. Available: <http://malwareint.blogspot.com/2010/02/spyeye-bot-part-two-conversations-with.html>
- [39] G. Bonfa, “Fast Overview of SpyEye,” Feb. 2010. [Online]. Available: <http://evilcodecave.blogspot.com/2010/02/fast-overview-of-spyeye.html>
- [40] Malware Protection Center, “Research Trojan:Win32/Spyeye,” Malware Protection Center: Threat Research and Response, Tech. Rep., Mar. 2010. [Online]. Available: <http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Trojan:Win32/Spyeye>

VITA

Joshua Michael Eads was born on June 8, 1987 to James and Deborah Eads of Independence, Missouri. He graduated from Lee's Summit High School in May of 2005 and enrolled as an undergraduate in Computer Science at the University of Missouri - Rolla (now Missouri University of Science and Technology). While an undergrad, he worked internships at Hy-Vee, Inc. in IT Support, Cerner Corp. in Radiology Software Development and Sandia National Laboratories in the Center for Cyber Defenders. During his time at Missouri S&T, Joshua was active in the local Association for Computing Machinery chapter and served as its Secretary, Vice-President, and President - helping to increase the membership and financial independence of the local chapter.

Joshua graduated with a Bachelor of Science in Computer Science from the Missouri University of Science and Technology at the end of the fall semester in 2008. He then enrolled in the Master's degree graduate program in Computer Science at Missouri S&T and joined the Trustworthy Systems Lab run by Dr. Miller, Cynthia Tang Missouri Distinguished Professor of Computer Engineering. Joshua continued his interest in software development project management from his undergraduate program and served two semesters as a graduate teaching assistant for the Software Systems Development capstone course. As a graduate student, Joshua stayed active with ACM and created a new special interest group for reverse code engineering. Joshua received his Master's of Science degree in Computer Science in May 2010 and plans to continue employment at Sandia National Laboratories full-time starting in June 2010.