WiLDCAT: AN INTEGRATED STEALTH ENVIRONMENT FOR DYNAMIC MALWARE ANALYSIS

by

AMIT VASUDEVAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2007

To my loving wife Deepa, my dearest son Arjun, my dear brother, parents and in-laws

ACKNOWLEDGEMENTS

I thank OmSakthi (the Almighty) foremost. Without Her I would not be what I am today. I am extremely thankful to my loving wife Deepa and my dearest son Arjun for their endless love, sacrifice, endurance and encouragement without which this work would not have been possible. I deeply thank my parents Srinivasan Vasudevan and Jayalakshmi Vasudevan, my brother Ajit Vasudevan and my in-laws Prof. V. Murali and Dr. V Murali for their love, support and encouragement in every walk of my life.

I would like to express my sincere gratitude to my advisor Ramesh Yerraballi for constantly motivating and encouraging me and for providing me great guidance and support during the course of this research work. I would like to thank Professors Roger Walker, Farhad Kamangar, Gergely V. Zaruba, Manfred Huber and David Levine for their interest in my research, for taking time to serve in my dissertation committee and for their comments, suggestions and guidance.

I would like to thank Bob Weems for all his support during the years I have been in UTA as a graduate teaching assistant and an instructor. I would also like to acknowledge the support from the Computer Science and Engineering Department at UTA for providing me with fellowships and assistantships at the time of need. Finally, I am also grateful to all the professors and mentors who have helped me throughout my career, both in India and United States.

April 13, 2007

ABSTRACT

WiLDCAT: AN INTEGRATED STEALTH ENVIRONMENT FOR DYNAMIC MALWARE ANALYSIS

Publication No. _____

AMIT VASUDEVAN, Ph.D.

The University of Texas at Arlington, 2007

Supervising Professor: Ramesh Yerraballi

Malware a term that refers to viruses, trojans, worms, spyware or any form of malicious code is widespread today. Given the devastating effects that malware have on the computing world, detecting and countering malware is an important goal. Malware analysis is a challenging and multi-step process providing insight into malware structure and functionality, facilitating the development of an antidote. To successfully detect and counter malware, malware analysts must be able to analyze them in binary, in both a coarse- (behavioral) and fine-grained (structural) fashion. However, current research in coarse- and fine-grained code analysis (categorized into static and dynamic) have severe shortcomings in the context of malware. Static approaches have been tailored towards malware and allow exhaustive fine-grained malicious code analysis, but lack support for self-modifying code, have limitations related to code-obfuscations and face the undecidability problem. Given that most if not all recent malware employ self-modifying code and code-obfuscations, poses the need to analyze them at runtime using dynamic approaches. Current dynamic approaches for coarse- and fine-grained code

analysis are not tailored specifically towards malware and lack support for multithreading, self-modifying/self-checking (SM-SC) code and are easily detected and countered by ever-evolving anti-analysis tricks employed by malware.

To address this problem, we propose WiLDCAT, an integrated dynamic malware analysis environment that facilitates the analysis and combat of malware, that are ever-evolving, becoming evasive and increasingly hard to analyze. WiLDCAT cannot be detected or countered in any fashion and incorporates novel, patent pending strategies for both dynamic coarse- and fine-grained binary code analysis, while remaining completely stealth. The environment allows comprehensive analysis of malware code-streams while selectively isolating them from other code-streams in real-time. WiLDCAT is portable, efficient and easy-to-use supporting multithreading, SM-SC code and any form of code obfuscations in both user and kernel-mode on commodity operating systems. It advances the state of the art in research pertaining to malware analysis by providing the toolkit that was sorely missing in the arsenal of malware analysts, until now!

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A malware is a program that can affect, or let other programs affect, the confidentiality, integrity, the data and control flow and the functionality of a system without explicit knowledge and consent of the user [6]. This includes viruses, trojans, worms, spywares or any form of malicious code. A classification of malware according to its propagation method and goal can be found in [64]. Given the fact that malware is widespread today and knowing the devastating effects malware can have in the computing world, detecting and countering malware is an important goal.

1.1 Malware Analysis

To successfully detect and counter malware, one must be able to analyze them in both a coarse and fine grained fashion, a complex and challenging process termed malware analysis. Malware analysis helps in gaining insight into malware structure and functionality facilitating the development of an antidote. Consequently, malware analysis environments (sandboxes) require the aid of various coarse and fine grained analysis tools.

The first and a very important step in the analysis process, involves monitoring malware behavior with respect to the system, to construct an execution model of the malware which aids in further finer investigations. As examples, the W32/MyDoom [44], W32/Ratos [74] and their variants propagate via e-mail and download and launch external programs using the network and registry. Such behavior, which includes the nature of information exchanged over the network, the registry keys used, the processes and files created etc., is inferred by employing coarse grained analysis on malware specific code and

related services of the host operating system (OS) pertaining to process, network, registry, file etc. Once such behavior is known, fine grained analysis tools such as debuggers and code tracers are employed on the identified areas to reveal finer details such as the polymorphic layers of the trojan, its data encryption and decryption engine, its memory layout etc.

Fine-grained malware analysis is a challenging task that provides important pieces of information that are key to building a blueprint of the malware core structure and functioning, that aids in the detecting and countering the malware and its variants. As an example, the W32/MyDoom trojan with variants commonly known as W32/MyDoom.X-MM (where X can be A, B, R, S, G etc.) share the same metamorphic code layers, encryption/decryption engine and similar anti-analysis schemes. The Netsky [20], Beagle [29] and Sobig [42] worms are some other examples of how malware are coded in an iterative fashion to add more features while retaining their core structure. Thus, once the core structure of a malware is documented, it becomes easy to tackle the rest of its variants as well as other malware which share a similar structure. Also, with malware writers employing more complex and hard to analyze techniques, there is need to perform coarse- and fine-grained analysis of malicious code to counter them effectively.

1.2 Research Motivations

Current research in malware analysis can be broadly categorized into static and dynamic approaches. Static approaches allow exhaustive fine-grained analysis because they are not bound to a specific execution instance. They allow detection of malicious code without actually running the program, ensuring that the malices discovered will never be executed and incur no runtime overhead. In spite of such powerful properties, static analysis has some limitations. With static analysis there is the problem that the analyzed code need not be the one that is actually run; some changes could be made

between analysis and execution. This is particularly true with polymorphism [86, 69, 73] and metamorphism [73] that are techniques employed by most if not all current generation malware. Also it is impossible to statically analyze certain situations due to undecidability (eg. indirect branches). Further, static code analysis also has limitations related to code obfuscation, a technique used by malware to prevent their analysis and detection.

Dynamic approaches overcome these limitations by analyzing the code during runtime, ensuring that the the analyzed code is the one that is actually run without any further alterations. Dynamic approaches can further be categorized into coarse- and fine-grained. Though there has been extensive research on static malware analysis, not much has been done in the area of dynamic malware analysis. Current coarse-grained dynamic malware analysis tools [26, 28, 66, 55] make use of probe-based instrumentation used for normal programs. The basic idea is to insert code contructs at desired location for monitoring purposes. Certain general purpose system tools are also used for coarse-grained malware analysis such as FileMon [61], Regmon [62], PortMon [59] etc. These use adhoc methods such as system call rehousing and filter device drivers. Currently dynamic fine-grained malware analysis can be achieved by employing debuggers and/or fine-grained instrumentation frameworks. When using a debugger such as Softice [25], WinDBG [56] etc., the basic approach is to set breakpoints on identified areas and then trace the desired code-stream one instruction at a time to glean further information. Alternatively one could also employ a fine-grained instrumentation framework such as Pin [41], DynamoRIO [11] etc., for automated tracing of code-streams. The strategies employed in the above methods are not equipped to handle malicious code effectively and have severe shortcomings in the context of current (and future) generation malware.

Probe-based methods fail in the context of current generation malware. Probe-based methods are not transparent because original instructions in memory are over-

written by trampolines. Most if not all current generation malware are sensitive to code modification and even instrument certain OS services for their functioning and stealthness. Also recent trends in malware show increasing anti-analysis methods, rendering probe-based schemes severely limited in their use [80]. Techniques used in general purpose tools used in the analysis process such as system call rehousing and filter device drivers are only limited to a subset of the OS functions and cannot be applied to malware code-streams. Further there are anti-analysis schemes employed against such methods in recent malware [72, 71, 70].

Current debugging and fine-grained instrumentation techniques can be easily detected and countered by the executing malware code-streams. Many malware employ techniques such as code execution timing, where the malware time their executing code thereby easily detecting that they are being analyzed (since debugging and/or automated code tracing incur latency that is absent during normal execution). A few examples are W32/HIV [46], W32/MyDoom [44], W32/Ratos [74], and their variants. Further, such malware contain ad-hoc detection schemes against popular debuggers such as Softice, WinDBG, etc. Current debugging and fine-grained instrumentation techniques do not carry support for self-modifying and/or self-checking (SM-SC) code. Most if not all malware are sensitive to code modification, employing subtle anti-analysis techniques and code obfuscations that defeat breakpoints in debugging and the process of automated code tracing using fine-grained instrumentation frameworks. For example, W32/MyDoom and W32/Ratos employ integrity checking of their code-streams with program counter relative code modification schemes which render software breakpoints and current fine-grained instrumentation frameworks unusable.

Malware that execute in kernel-mode are even tougher to analyze using current dynamic fine-grained techniques, since they have no barriers in terms of what they can access. For example, W32/Ratos employs a multithreaded polymorphic/metamorphic

code engine running in kernel-mode, and overwrites the interrupt descriptor table (IDT) with values pointing to its own handlers. Current fine-grained instrumentation frameworks do not handle kernel-mode code and do not carry adequate support for multithreading. Current debugging techniques provide kernel-mode code support but do not support multithreading in kernel-mode. Furthermore, recent trend in malware has been to employ debugging mechanisms supported by the underlying processor within their own code, thereby effectively preventing analysis of their code using current debugging techniques. Examples include W32/Ratos, which employs the single-step handler (used for code tracing) to handle its decryption and W32/HIV which uses debug registers (used for hardware breakpoints) for its internal computation.

This situation calls for a integrated environment, constructed with techniques specifically tailored for current and future generation malware that empowers a malware analyst to dynamically analyze (in both a coarse- and fine-grained fashion) and counter malware that are becoming evasive and increasingly hard to analyze.

1.3 Summary of Contributions

In this dissertation, we present novel and powerful techniques for both coarse and fine-grained dynamic malware analysis. In short, during the course of this research, we have developed a mechanism for stealth stalling of executing code-streams at runtime, we have developed techniques for stealth malware behavior analysis, we have developed strategies for stealth structural analysis of malware code-streams, we have developed a mechanism that facilitates implementation of the above techniques on commodity OSs without sources. WiLDCAT is a comprehensive toolkit that realizes the individual concepts implementing a stealth environment for dynamic analysis of malware. We also describe our experiences in analyzing various next generation malware using WiLDCAT.

Below, we summarize our contributions based on our motivation provided in the previous section:

- Stealth Stalling of Code-streams: We propose radical techniques to stall execution of binary code-streams at runtime in a completely stealth fashion. Our stealth breakpoint framework codenamed VAMPiRE provides breakpoint ability for code,data and I/O in both user and kernel-mode on commodity OSs without any form of modification to the executing code-streams. We have carried out performance studies using VAMPiRE and have shown that its latency is well within limits to suit interactive analysis and in some cases even comparable to existing hardware and software based approaches of code stalling. We have published our techniques and their performance in [78].
- Coarse-grained Malware Analysis: We propose novel code analysis techniques for coarse-grained malware analysis. SPiKE, our coarse grained instrumentation framework provides the ability to monitor and/or alter semantics of code constructs at runtime in a completely stealth fashion. The key feature of the framework is its persistent nature, in the sense that it allows an executing code stream to deploy its own instrumentation over SPiKE, while still ensuring that SPiKE is the first to get control. We have comprehensively evaluated SPiKE by comparing it to existing instrumentation strategies both in a qualitative and quantitative fashion. Our qualitative comparison shows that the framework is the first of its kind in tailoring an instrumnetation strategy for malware analysis while our quantitative comparsion shows that the framework is efficient enough for interactive monitoring and/or analysis. We have published our techniques and their performance in [80, 76].
- Fine-grained Malware Analysis: We propose novel techniques that extend the current virtual machine technology for fine-grained analysis of malware code-streams. Cobra, our stealth-localized execution framework provides an on demand, selective,

localized and stealth fine-grained execution environment in both user- and kernel-mode on commodity OSs. Cobra can analyze the execution of a code-stream from as fine as an instruction level to a group of instructions. The framework can do so while allowing other code streams to execute as is. Cobra can execute privilege level code and allows manipulation of critical system structures while ensuring complete control using the mechanism of cloning. We have carried out performance studies using Cobra and have shown that the latency is within limits to suit interactive analysis. We have published our techniques and their performance in [79].

- Dynamic Extensions: We propose retargetable techniques to dynamically monitor and/or enhance parts of the host OS without access to sources. We also propose techniques to load portions of the environment in the context of a specific process/thread under the host OS in a dynamic fashion. Our framework code-named Sakthi, enables binary instrumentation across multiple operating systems and machine architectures. It is based on the concept of dynamic extensions and provides a code injection mechanism, which lends itself to implementation on a wide range of commercial and free operating systems. Additionally our framework is capable of providing options for global and local extensions, is able to extend pure or arbitrary OS constructs and deploy code in a new or an already executing process within the OS with minimal downtime. We have published our techniques and results in [77].
- Experiences: We have used WiLDCAT to analyze various malware such as W32/HIV, W32/Ratos, W32/MyDoom and Troj/Feutel-S which are are examples of the state-of-the-art in current generation malware. These malware employ sophisticated techniques that thwart existing code analysis strategies and can be expected to become de-facto in the near future if not more advanced. However, with WiLDCAT we analyze these malware in a comprehensive fashion documenting both their

behavior as well as structural organization thereby showing the impact and efficacy of our proposed techniques.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows: We start by discussing the characteristics of the next generation malware in Chapter 2. We follow this by a discussion on current work related to code analysis techniques including those that apply one or more of such techniques for malware analysis in Chapter 3. In Chapter 4 we present an overview of WiLDCAT, its design principles, features and components. We describe our techniques for stealth stalling of code streams in Chapter 5. We follow that with a detailed discussion of our techniques for coarse- and fine-grained malware analysis in Chapter 6 and Chapter 7 respectively. We then discuss our techniques for extending parts of commodity OS in Chapter 8. In Chapter 9 we discuss miscellaneous stealth techniques that WiLDCAT employs alongwith how the environment supports the next generation of malware code-streams. We describe our experiences with WiLDCAT in analyzing several real-world malware in Chapter 10 thereby showing the impact and utility of the environment. Finally, we conclude and provide future work in Chapter 11.

CHAPTER 2

NEXT GENERATION MALWARE TRAITS

To devise strategies for malware analysis, one needs to understand the facets of a malware code-stream. This chapter discusses the features that embody the next generation malware, a subset of which applies to current generation malware as well.

2.1 Self-Modification

Most if not all malware code-streams employ techniques that change their executing code-streams for different deployments and even change their code during run-time for a particular deployment. This is referred to as self-modification and helps in preventing detection and analysis since the code-streams are never constant. The following sections describe various techniques for self-modification in further detail. A point to be noted is that a malware can employ one or more of the following schemes within its code-streams.

2.2 Encryption

A malware code-stream can encrypt parts of itself, which is then decrypted during runtime. This decryption is carried out by a decryption code which remains constant. This scheme ensures that one cannot employ a specific signature to identify the encrypted code. However, since the decryption code is constant, it call still be identified and further once the decryption algorithm is unveiled after the study of the decryption code it is easy to decrypt the encrypted malware code-streams.

2.3 Oligomorphism

With oligomorphism, the malware code streams employ various encryption algorithms and use different decryption code at different deployments for decryption purposes. This entails the analysis of decryption code for various deployments in order to decrypt the actual malware code-streams.

2.3.1 Polymorphism

With polymorphism, a malware code-stream can rearrange portions of itself while still maintaining its functionality. This rearrangement can be done multiple times during a single deployment and usually varies among multiple deployments. This makes it more diffcult to identify pertinent code sections for analysis purposes.

2.3.2 Metamorphism

With metamorphism this rearrangement is taken to an instruction level where parts of a code-stream have their instructions rearranged in such a fashion that they achieve the same functionality.

2.3.3 Z-morphism

We envison that the next generation malware are headed towards employing a technique where they change both their functionality as well as the rearrangement of parts of their code-streams thereby truly mutating into a new variant. We call this Z-morphism.

2.3.4 VM-morphism

We envison that a malware code-stream can also employ a virtual machine approach to its execution. In other words, instead of executing real instructions on the host

processor, the malware code-streams execute instructions on a virtual processor. Further, this virtual processor can change during a single deployment and in between deployments making it very difficult to analyze them.

2.4 Self-Checking

Self-checking refers to the mechanism by which a malware code-stream scans itself to detect any form of modification done to it. This might be the case for example when a software breakpoint or a probe-based instrumentation is inserted on the code stream.

2.5 Obfuscation

Obfuscation is a technique by which the malware code-streams make it difficult for the analysts to understand a section of their code. The basic idea is to encode instructions such that they do not appear one after the other, instead some instructions are coded carefully so that they overlap in memory. This also makes it difficult to insert any form of code stalling mechanism as it might be difficult to locate a particular instruction within the obfuscated code stream.

2.6 Multithreaded Envelopes

mulithreaded coding techniques use more than a single code-stream for malware functionality. For example on the w32/ratos there are several kernel-mode threads that are launched. one of these threads continuously monitors the system table for any changes to pointers while a group of threads are involved in decryption of the actual payload.

2.7 Stealth Techniques

Malware code-streams can employ stealth techniques in order to prevent any portion of them being detected or analyzed. A malware code-stream can employ stealth techniques on:

- Disk: A malware can mark sectors used for its functioning as BAD to prevent the host OS from seeing the data. Malware can also store data at the end of a disk. This area of the disk is generally not used for data storage. Some malware can also format a disk beyond their capacity and store their data within those sectors. Further details can be found in [73].
- Filesystem: A malware can hide its critical files from the host OS and analysis tools by instrumenting portions of the host OS responsible for file I/O. As an example the W32/Ratos instruments the ZwQueryDirectoryInformation API within the Windows kernel in order to prevent any of its critical files being seen.
- Memory: A malware can also employ stealth within its code-streams by executing only portion of its code within a windowed memory region, thereby exposing only a small fraction of itself at any given time. As an example the W32/Ratos uses various memory allocations within both user- and kernel-mode code-streams where a part of the code-stream is decrypted, executed and then destroyed by deallocation of the memory areas.

2.8 Use current analysis aids for functioning

A malware can employ analysis aids used by analysis tools for their own functions. As an example, W32/HIV uses debug registers (which are used to trigger hardware breakpoints on the IA-32 and compatible processors) within its code-streams for its internal computations. W32/Ratos uses the single-step interrupt (used for single-stepping

instructions during analysis) for its decryption purposes. W32/MyDoom uses instrumentation (used for monitoring) to gain control at desired portions of the host OS to conceal its critical resources and semantics.

2.9 Privileged and Unprivileged Execution

A malware can employ both user- as well as kernel-mode code-streams for its functionality. While user-mode code-streams are restricted in performing certain critical modifications, kernel mode code-streams have no restrictions in terms of the data they can access. This makes such malware code-streams very difficult to analyze without putting the system into an unstable state. As an example W32/Ratos uses user-mode code-streams for a bulk of its functionality while employing anti-analysis techniques and decryption within kernel-mode.

2.10 Miscellaneous Anti-Analysis Tricks

This section discusses miscellaneous tricks that malware code-streams can employ to prevent their analysis.

- Checking hardware/software breakpoints: A malware code-stream can check for any active hardware breakpoints by examining certain processor specific registers. For example, on the IA-32 and compatible processors, a malware could check for the DR7 debug control register to detect any active hardware breakpoint. Software breakpoints can be easily detected with self-checking as discussed previously. Eg. W32/HIV, W32/Ratos
- Checking instrumentation: A malware code-stream can check for any active instrumentation of host OS, supporting libraries or the malware specific libraries. For example, probe-based instrumentation can be checked by comparing the first few

bytes of a known target function with the original code for that function to detect a probe. Similarly, table-based instrumentation can be detected by comparing the addresses to default addresses for the host OS.

- Timing check: A malware code-stream can also employ relative timing of its executing code. This technique can be used to detect any form of code tracing such as single-stepping or VM-based approaches. Timing checks can also be used for regular breakpoints since the breakpoint activation will result in increased latency that is not normally present.
- Privilege level check: A malware code-stream can also check for its executing code segment privilege in order to determine if it is being run inside a VM. A malware executing in kernel-mode when run inside a VM will have its privilege level restricted. Thus, a check to the code segment privilege will unveil the VM being employed.
- Checking interrupt handlers via tunneling: A malware can use VM based approach in order to execute interrupt handlers used for breakpoints in order to determine if it is a standard OS handler or not. If a debugger employs hardware breakpoints, it will need to override the existing handler. Thus, with this tunneling approach the addresses of execution will be outside of the standard OS code which is an indication of an analysis tool being employed.
- Trapping Keyboard/Mouse: Since code analyis involves the keyboard (for interacting with the analysis tool and for tracing the code), a malware code-stream could do direct I/O with the keyboard controller in kernel-mode, disabling the keyboard altogether while its kernel-mode code-stream are executing.
- Stack based: Single-stepping causes certain extra information being stored on the stack of the executing code-streams. A malware code-stream can use this fact to

detect extra information on its stack that would otherwise not be present during normal execution.

- Checking for specific signatures: A malware code-stream could check for a known analysis tool by either looking at the list of drivers within the system, the registry/configuration entries and tool specific signatures. For example W32/Ratos checks for popular debuggers such as WinDBG and Softice by looking for their drivers and modified registry entries.
- checking for OS specific structures/APIs: A malware code-stream could also employ OS specific structures to detect if its being analyzed. For example on the Windows OS, the IsDebuggerPresent API returns true if a debugger is currently executing. Also the Thread Information Block contains certain values that are different when a debugger is active.
- Detecting emulators: A malware code-stream can employ processor specific instructions such as MMX, coprocessor or undocumented CPU instructions and processor microcode specific behavior (prefetch queue, branch prediction etc.) to detect emulators. Many emulators do not handle all the possible instructions and the processor microcode behavior and hence can be detected.

CHAPTER 3

RELATED WORK

This chapter highlights various topics of research that will provide background for the underpinnings of this research. Current research in the area of malware analysis can be broadly categorized into static and dynamic approaches [6]. Both methods have their advantages and disadvantages and are complimentary. Static approaches to malware analysis can be used first, and information that cannot be gleaned statically can then be dynamically ascertained.

3.1 Static Analysis

Static approaches to malware analysis extend techniques related to verifying security properties of software at a source level [1, 8, 14, 15, 32, 39] to binary (since for a malware, in most if not all cases, there is no source-code availability). These approaches are used primarily for fine-grained malicious code analysis. Bergerson et. al [7, 5] present techniques that disassemble the binary and pass it through a series of transformations that aid in getting a high-level imperative representation of the code. The binary is then sliced to extract the code fragments critical from the standpoint of security and malicious code. Giffin et al [27] disassemble a binary to remotely detect manipulated system calls in a malware. Many malware detection techniques are based on static analysis of executables. Kruegel et al. [34] employ static binary analysis to detect kernel-level rootkits. SAFE [17] and Semantic-Aware Algorithm [19] are other examples of malware detection algorithms employing similar static analysis techniques. Static approaches allow exhaustive fine-grained analysis because they are not bound to a specific execution

instance. They enable detection of malicious code without actually running the program. Therefore, the malices discovered will never be executed. On the performance side, there is no run-time overhead associated with a static analysis. After just one analysis, the program can run freely. In spite of these beneficial properties, there are some limitations. The main problem with static code analysis is that the analyzed code need not be the one that is actually run; some changes could be made between analysis and execution. This is particularly true in self-modifying techniques such as polymorphism [86, 69, 73] and metamorphism [73] that are ubiquitous in most malware code streams today. Static approaches also have limitations related to code obfuscation [23, 24, 84]. They employ a dissassembler as an essential step in their analysis procedure. Linn and Debray [38] and Christodorescu and Jha [18] demonstrate that simple obfuscations can thwart the dissassembly process. While Kruegel et al [33] present techniques for disassembling obfuscated executables, they are unable to handle situations such as indirect obfuscation [53, 82], instruction overlap [22] etc.

3.2 Dynamic Analysis

Dynamic approaches analyze malware during runtime. Though dynamic approaches incur a runtime overhead and are not exhaustive in terms of code coverage, they overcome the main limitation of static approaches in ensuring that the analyzed code is the one that is actually run without any further alteration. Thus dynamic approaches can handle SM=SC code or any form of code obfuscations. Dynamic approaches can be categorized into coarse-grained and fine-grained.

3.2.1 Coarse-grained

Instrumentation – the ability to control constructs pertaining to any code – is a technique that is used for dynamic coarse- and fine-grained analysis. Constructs can

either be pure or arbitrary. Pure constructs are code blocks following a standard calling convention (e.g OS APIs, standard library functions etc.). Arbitrary constructs on the other hand are code blocks not adhering to a standard calling convention (malware specific functions, OS internal functions etc.). Control means access to a construct for purposes of monitoring execution or possible semantic alteration. As an example, if one considers the W32/Ratos, a trojan running under the Windows OS, the ExAllocatePool OS API (a pure construct) can be instrumented with a change in semantic, to ensure that memory pages allocated using the API by the malware are setup such that any access to such memory pages in terms of reads, writes and/or executes can be tracked. This allows documenting the memory usage in kernel-mode and the metamorphic code envelopes of the trojan. On the other hand, a part of the trojan internal function for decryption (an arbitrary construct) can be instrumented to monitor memory regions where the malware executes obfuscated code for its functionality.

There are various research related to the area of instrumentation. Broadly, instrumentation can be categorized into source level or binary level. Source level instrumentation includes insertion of wrappers in the program source code [9], which transfer control to the instrument during program execution. While source level instrumentation can be applied to OSs and applications that are open source, the task becomes complicated with deployments that are commercial and binary only. Further, in the context of malware analysis, most if not all malware exist in the form of binaries. To this end, binary level instrumentation accomplish instrumentation without the necessity of source code. Current research in binary instrumentation can be categorized into the following:

3.2.1.1 Offline binary instrumentation

Offline binary instrumentation involves rewriting the program binary to insert instrumentation constructs. This art was pioneered by Atom [68], followed by others such

as EEL [36], Etch [57] and Morph [87]. Offline approaches have a serious drawback in that, the tool may not have enough information to deal with mixed code and data within the executable. In the context of malware binaries, offline approaches do not suffice as most if not all malware are sensitive to code modification, being self-modifying and or self-checking. Other difficulties with offline systems are indirect branches, dynamic libraries and dynamically generated code.

3.2.1.2 Online Probe-based binary instrumentation

Online Probe-based binary instrumentation works by writing a control transfer instruction (CTI) at the destination memory address corresponding to the construct being instrumented during runtime. The CTI is usually a CALL, JUMP or TRAP instruction and is responsible for invoking the instrument when the original construct is invoked by a target code-stream. Detours [30], DynInst [12], Vulcan [67], Dtrace [13] and DProbes [51] are examples of some popular probe-based dynamic binary instrumentation frameworks.

Some of the above frameworks such as Detours add more flexibility by allowing the instrument to execute the orignal function within themselves. The basic idea involves disassembling the instructions at the original function into a temporary code area and appending a CTI at the end to branch to the instruction following the CTI that is written at the start of the original function. When control is transferred to the instrument, it can invoke the original function via the temporary code area.

Most of the current research in/or related to coarse-grained malware analysis employ probe-based instrumentation for their functionality. DaMon [26] is a dynamic monitoring system uses a similar technique to dynamically enforce a security policy to stop certain malicious actions on resources such as ports, registry, processes etc. Most host-based intrusion detection systems such as Bro [55] and StJude [10] and trojan and spy-

ware monitors such as Anti-Trojan Shield [2] and Microsoft Anti-Spy [50] also hinge on probe-based dynamic binary instrumentation for their functioning.

There are various drawbacks to probe-based binary instrumentation. In the context of malware, it cannot be used to instrument malware specific functions since most if not all malware are very sensitve to code modifications. With recent trend in malware showing increasing anti-analysis schemes, they can be easily detected and countered. Other problems with probe-based approaches are related to arbitrary construct instrumentation on architectures where instruction sizes vary (i.e x86). In such cases, an instruction cannot be replaced with one which is greater than its size, since it would overwrite the following instruction.

3.2.1.3 JIT-based

JIT-based binary instrumentation works by running the program code inside a VM. The VM is usually based on dynamic compilation where blocks of instructions are executed at a single time for efficiency. Coarse-grained instrumentation is achieved, when an instruction inside the VM transfers control to a memory-address that is being instrumented.

Pin [41], DynamoRIO [11], Valgrind [52], Diota [43] and Strata [63] are examples of some popular JIT-based dynamic binary instrumentation frameworks.

Execution of code inside a virtual machine can result in the betrayal of the real-state of the program and prevent malicious code from executing as on a real system. Current JIT-based binary instrumentation frameworks are not tailored towards malware and hence do not carry adequate support for multithreading and do not support SM-SC code. Further, they do not support kernel-mode execution and can be easily detected. Also, JIT-based binary instrumentation is very slow when compared to probe-based binary instrumentation for coarse-grained instrumentation.

3.2.1.4 Table-based

When a user-mode component makes a privileged system call, control is usually transfered to a software interrupt handler in the host OS kernel. This handler takes a system call number that indexes into a system service table to find the address of the function that will handle the request. By replacing entries in this table with addresses of instruments, it is possible to intercept and replace, augment, or monitor system services. We call this method as Table-based binary instrumentation. As an example, on the Windows OS, KiSystemServiceDispatcherTable is the system table containing pointers to various kernel functions which are invoked via INT 2Eh by the user-mode library NTDLL.

Janus [28] provides a secure environment to execute untrusted applications. It intercepts and filters dangerous system calls under Solaris to reduce the risk of a security breach by restricting the program's access to the operating system. Regmon [62] and ProcMon [58], general purpose tools for monitoring real time registry and process creations under the Windows OSs also use table-based binary instrumentation, instrumenting registry and process specific system calls. STrace [47] and Linux Trace Toolkit [85] are some more examples of tools which use this method.

Table-based binary instrumentation suffers from the fact that it is not comprehensive. Only system calls indexed by a system table can be instrumented and not constructs from other portions such as shared libraries or malware specific code-streams. Further, it relies on the appropriate index for the system call which can change between different versions of the OS. Also, recent malware such as the W32/MyDoom [44] have shown anti-analysis methods against this scheme. The replaced addresses in the system table are outside of the image space of the host OS kernel, which is detected and countered by the malware. (see Chapter 10 for more information on the W32/MyDoom).

3.2.1.5 Filter-based

Filter-based binary instrumentation works by using filter device drivers in the host OS [54]. Filter device drivers allow a driver to intercept and alter parameters before the actual interfaces for the device is invoked. As an example, if one considers the ZwCreateFile API under the Windows OS, it finally translates into I/O calls to the file system driver. To instrument ZwCreateFile, a filter driver can be used for the file system driver. Thus, the filter driver will be the first to receive all I/O requests to the file system driver. The filter driver can then determine if it encounters a create I/O call and in such a case invoke the instrument. The instrument can then access the parameters associated with the call, alter them if needed and pass the request down to the original driver.

FileMon [61], a general purpose tool to monitor file system activity on the Windows OS works by employing a file system filter driver. PortMon [59] is another tool which captures realtime access to I/O ports on the system. It does so by using a serial/parallel port filter driver. Many intrusion detection systems such as NetTop [48] and tools such as TDIMon [60] use network filter drivers to monitor network traffic.

Filter-based binary instrumentation has several shortcomings in general as well as in the context of malware. Filter drivers can only be employed on device objects and hence cannot be used to instrument general purpose constructs pertaining to processes, memory, registry etc. Futher, instrumentation via filter drivers are not very flexible. As an example, an instrument for a CreateFile API might wish to invoke the original API one or more times if it is implementing a proxy. However, if the instrument is via a filter driver, there is no way to traverse the driver stack multiple times. Also, filter drivers are easy prey to detection. W32/Mydoom, W32/Lamin, W32/Kipis and Backdoor/DkAngel are a few of the many malware that detect tools such as FileMon by the presence of their filter drivers.

3.2.1.6 Hardware-based

Hardware-based binary instrumentation work by employing processor breakpoint support for instrumentation purposes. Breakpoints are debugging aids that help in stalling the execution of code at runtime. Hardware breakpoints are supported by the underlying processor and support precise breakpoints on code, data and I/O without any modifications to the target code-stream. They are deployed by programming specific processor registers to specify the breakpoint locations and type. As an example, the IA-32 (and compatible) processors use the debug registers to set hardware breakpoints [31].

SoftIce [25], WinDBG [56] and GDB [40] are popular debuggers that make use of hardware-based binary instrumentation to gain control at specified execution points in both user- and kernel-mode.

Hardware breakpoints do not involve any form of code modification and support SM-SC code and any form of code-obfuscations in both user and kernel-mode. However, current processors provide a very limited number of hardware breakpoints (only 2-4 locations). Thus, a serious restriction is imposed to instrument a desired number of constructs. Further, there are techniques being employed in recent malware that prevent the use of hardware breakpoints to analyze them. For example, the W32/MyDoom and the W32/Ratos use processor debug registers and breakpoint exception for their internal computations.

3.2.2 Fine-grained

Dynamic fine-grained malware analysis is an unexplored area where not much has been published. Cohen [21] and Chess-White [16] propose a virus detection model that executes in a sandbox. However, their model is not generic and does not allow fine-grained analysis at a level that can be used to document the internal workings of a malware.

Debuggers such as Softice [25], WinDBG [56], GDB [40] etc. enable dynamic fine-grained analysis in both user- and kernel-mode. Though current debuggers to some extent, support self-modifying code and code obfuscations, they are not tailored specifically towards malware analysis and fall prey to several anti-debugging tricks employed by them [78]. While code analysis using a debugger is manual (one has to trace instructions manually), tools such as Pin [41], Valgrind [52], DynamoRIO [11], Strata [63], Diota [43] etc. enable automated code tracing by employing a virtual machine approach. However, these tools are designed for normal program instrumentation and hence do not carry support for SM-SC code and code obfuscations. Further these tools do not carry adequate support for multithreading and cannot handle code in kernel-mode. Hypervisors such as VMWare [81], QEmu [4] etc. are able to handle multithreading in both user- and kernel-mode efficiently, but do not carry support for SM-SC code. Also, they are not tailored towards malware and can be detected and countered [73].

CHAPTER 4

OVERVIEW OF WiLDCAT

In this chapter we discuss the high level architecture of WiLDCAT. We discuss the core design principles that govern the environment architecture as well as various components that empower the environment with capabilities for both coarse- and fine-grained malware analysis

4.1 Architecture

The architecture of WiLDCAT is based on the premise that:

`The environment is the first to get control on a clean host`

From the premise, we are seeking a solution that will be 100% effective iff the environment is the first to be deployed on a clean host. We do not see this as a shortcoming since the environment is intended for use by malware analysts who can be expected to work on systems that are not infected in the first place. The two core principles governing the framework design are:

- Design for stealth: The framework should be completely invisible to the executing code-streams
- Design for malice: The framework must allow analysis of any form of malicious code at any privilege level of execution
- Design for self-resilience: The framework should not be used against itself.

From the first principle we are seeking a solution that is impossible to detect or circumvent in any fashion. From the second principle we are seeking a solution which can be applied to any current as well as future generation malware. From the third principle, the solution

must be such that it should allow a malware to use the same techniques as used by itself while still ensuring complete supervision during execution and analysis.

These principles have a substantial positive impact in that: it allows us to define an architecture that cannot be countered and is capable of analyzing not only current generation malware but also possible future generation.

WiLDCAT runs on commodity OSs in real-time, blending into the executing OS in a seamless and stealth fashion. The environment currently runs on Windows (9x, 2K and XP) and Linux on the IA-32 (and compatible) processors but is portable on any platform containing support for virtual memory. Figure 4.1 shows the architecture of WiLDCAT in its current version. As seen the core of the environment resides in the highest privilege level allowing the environment complete control over code-streams in any privileged level of execution. A point to note is that subsystems/tools that are still not mature and are under research and development are shown greyed in Figure 4.1.

4.2 Features

Following are the salient features of WiLDCAT:

- *Stealth:* WiLDCAT and the techniques employed by it are completely invisible to the executing code-streams. This is very important as many recent malware are showing increasing resistance against analysis employing various anti-analysis tricks.
- *Persistent:*Further, once WiLDCAT is deployed, the environment ensures that it is always the first to get supervision while allowing other code-streams to make full use of hardware or software debugging aids within themselves.
- *Conducive for malicious code execution:* The strategies employed by WiLDCAT are dynamic, completely transparent, supporting multithreading, SM-SC code and any form of code-obfuscation in both user- and kernel-mode.

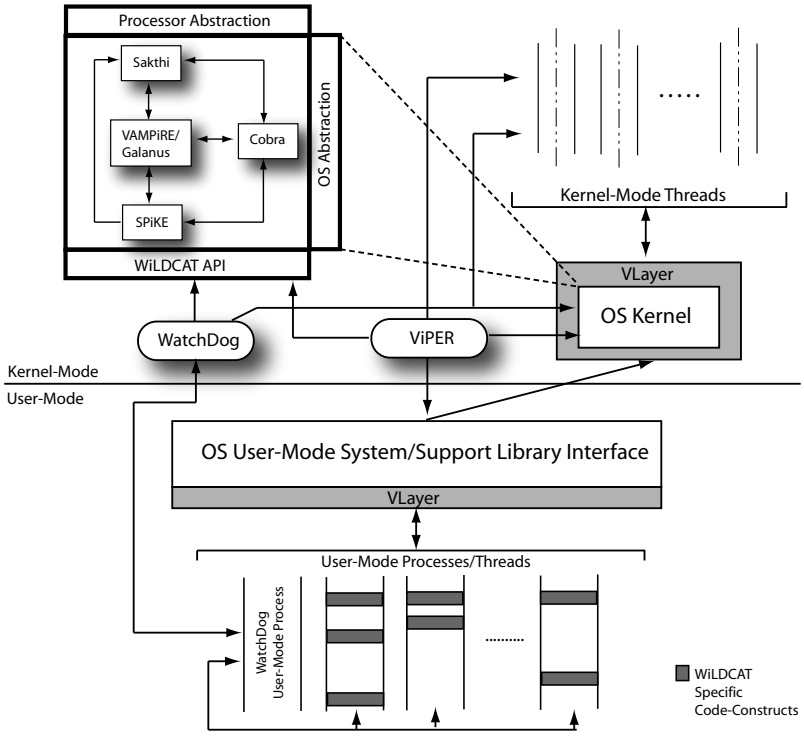


Figure 4.1. Architecture of WiLDCAT.

- *Comprehensive:* WiLDCAT allows a malware analyst to analyze code-streams dynamically in both a coarse- and fine-grained fashion.
- *Selective:* The environment supports selective isolation, whereby a malware specific code-stream can be isolated and analyzed while allowing other code-streams to execute as is.
- *Efficient:* WiLDCAT achieves its functionality without using an instruction level virtual machine. The environment uses techniques such as Stealth Breakpoints, Invisi-Drift, Redirection and Localized-Executions to achieve interactive response while ensuring complete control over executing code-streams.
- *Portable:* The strategies employed by WiLDCAT for both coarse- and fine-grained analysis hinge on the underlying virtual memory and are easy to implement on most platforms. The environment currently runs on Windows (9x, 2K & XP) and

Linux on the IA-32 (and compatible) processors and is portable on any platform supporting virtual memory.

- *Easy-to-use:* WiLDCAT allows an analyst to customize every aspect of the environment allowing them to tie actions to events in real-time. Further, the WiLDCAT SDK allows authoring of tools that are specific to a given task.

4.3 Environment Subsystems

WiLDCAT is composed of a host of subsystems facilitating both coarse- and fine-grained malicious code analysis in manual and/or automatic fashion in both user- and kernel-mode. The following are the notable subsystems/tools comprising WiLDCAT (shown in Figure 4.1):

- VAMPiRE/Galanus : Forms the backbone of WiLDCAT and allows for dynamic and stealth stalling of executing code-streams in real-time. This subsystem uses a novel strategy based on virtual memory to provide mechanisms to selectively stall code-streams on a particular instruction, access to particular regions of memory and/or I/O instructions in both user- and kernel-mode in an efficient manner.
- Sakthi: Employs portable techniques that enable dynamic extension of the executing OS and the ability to inject and execute environment specific code-constructs in the context of the whole OS or a specified processes/threads.
- SPiKE: Endows the environment with the ability to support coarse-grained analysis on desired code-streams. SPiKE allows a malware analyst to observe, log, analyze and even change the way in which a malware interacts with the system at a high-level in real-time. This subsystem uses novel techniques such as Invisi-Drift and Redirection to instrument code-constructs/data locations in memory for monitoring purposes.

- Cobra: Facilitates automated fine-grained malicious code analysis under WiLDCAT. This subsystem helps a malware analyst in gleaning important pieces of information about the malware internal structure as it executes and build a blueprint of its core. This blueprint can then be used for detecting and countering the malware. Cobra also allows dynamic blueprint matching, a mechanism which allows a malware analyst to determine if a malware core structure is a variant or shares a similar structure with a previously analyzed malware, so that detection and combat methods may be reused albeit with minor modifications.
- VLayer: This subsystem is responsible for virtualizing certain OS components such as filesystems, network shares etc. to allow a malware analyst to execute malicious code-streams that cause severe damage to the underyling OS, without having to worry about putting the system into an unstable state. The virtualization technique is based on SPiKE and allows a malware to deal with virtualized resources while maintaining their physical bindings as well as ensuring that such virtualization remain undetectable.
- ViPER: This is a stealth multithreaded debugger which allows fine-grained malicious code analysis in both user- and kernel-mode. ViPER relies on VAMPiRE/Galanus at its core, coupled with other interesting strategies that are completely resilient to any form of anti-debugging tricks. It does not rely on any form of hardware support for debugging and supports SM-SC code and any form of code obfuscation.
- WatchDog: This is a tool that makes use of the core subsystems of WiLDCAT and enables monitoring of various user- and kernel-mode components of the OS, offering insight into malware behavior with respect to the system alongwith support for fine-grained analysis on desired code-streams. WatchDog is a simple yet versatile tool demonstrating the power of WiLDCAT, that allows real-time monitoring of Windows APIs related to files, processes, registry, network, memory and various

other sections of the OS both in user- and kernel-mode. It enables complete monitoring of drivers and dynamic link libraries that are being loaded and unloaded in the system. WatchDog also allows a malware analyst to deploy automated fine-grained analysis on desired code-streams during an analysis session. Further, the tool also allows instrumenting the entry and exit points of the dynamic link libraries and drivers with support for critical data access monitoring (process/thread blocks, exception chains etc.), dependencies, per process filters and a host of other features which make it an indispensable tool for dynamic malware analysis in a system. The tool also enables logging of several important pieces of a function invocation such as the parameters, the return value, the location it was invoked from, the stack and memory contents etc. `WatchDog` also features what we call *selective logging*, whereby only invocations arising from a specified range of memory is monitored. This is a very useful technique that helps maintain the clarity of information that is logged. The tool also features a script based interface allowing users to add real-time actions to monitored events.

The following chapters discuss the environment subsystems in more detail.

CHAPTER 5

STEALTH EXECUTION STALLING

The key solution to the problem of dynamic malware analysis is the ability to stall execution of code at runtime in a completely stealth fashion. In this chapter we present novel techniques towards stealth stalling of code-streams at runtime and discuss the design implementation and performance of VAMPiRE, one of the key subsystems of WiLDCAT.

5.1 Introduction

One of the important aspects of code analysis in general is the ability to stop execution of code being debugged at an arbitrary point during runtime. This is achieved using breakpoints, which can be of two types: Hardware and Software. Hardware breakpoints, as the name suggests, are provided by the underlying processor and support precise breakpoints on code, data and I/O. They are deployed by programming specific processor registers to specify the breakpoint locations and type. Software breakpoints on the other hand are implemented by changing the code being debugged to trigger certain exceptions upon execution (usually a breakpoint exception).

Software breakpoints support unlimited number of breakpoint locations but suffer from the fact that they modify the target code at runtime. This is clearly unsuitable in the context of malware since most if not all malware possess SM-SC capabilities and are very sensitive to changes made to their code. For example, viruses such as `W32.HIV` [46], `W9x.CIH` [45], `W32.MyDoom` [44] etc. use polymorphic/metamorphic code envelopes and employ a host of integrity checks to detect any changes made to their internal code

fragments, to prevent their analysis. Hardware breakpoints on the other hand do not involve any form of code modification and, hence, are the most powerful tool in the stalling execution of code in the context of malware. Current processors , however, provide a very limited number of hardware breakpoints (typically 2–4 locations). Thus, a serious restriction is imposed on a debugger to set desired number of breakpoints without resorting to the limited alternative of software breakpoints. Also, with the ever evolving nature of malware, there are techniques being employed that prevent the use of hardware breakpoints to analyze them. For example, the `W32.HIV` virus uses the processor debug registers and the breakpoint exception for its internal computations, thereby effectively thwarting hardware breakpoints.

VAMPiRE , a stealth breakpoint framework overcomes the shortcomings of existing strategies related to code execution stalling by offering the best of both worlds in the sense of unlimited number of precise breakpoints on code, data and I/O which cannot be detected or countered. This is achieved by employing novel and subtle stealth techniques that involve virtual memory, single-stepping and task state segments [1] (for processors supporting legacy I/O) — features found in most new and old processor architectures.

While various ideas using virtual memory for breakpoint purposes have been explored in many debuggers [3, 40, 25], most if not all, allow only data read and/or write breakpoints. Also none of them are specifically tailored for malware analysis and their breakpoint implementation can be easily detected and defeated. To the best of our knowledge, VAMPiRE is the first to combine virtual memory, single-stepping, task state segments (TSS) and stealth techniques to provide a stealth and portable breakpoint framework highly conducive for malware analysis. By *stealth* we mean that the breakpoints inserted using VAMPiRE is completely invisible to the code being debugged. The

[1] Task State Segments (TSS) are used to store all the information the processor needs in order to manage a task. This includes the processor registers, stacks, the task's virtual memory mappings etc.

framework is portable on any platform (OS and processor architecture) that supports virtual memory and single-stepping. The framework performance is well within the limits to suit interactive analysis.

5.2 Overview of VAMPiRE

Breakpoints under VAMPiRE are realized through a combination of virtual memory, single-stepping, TSS (for applicable processors) and simple stealth techniques. The basic idea involves breakpoint triggering by manipulation of memory page attributes of the underlying virtual memory system (for code, data or memory-mapped I/O breakpoints) and I/O port access control bits in the TSS (for legacy I/O breakpoints). Note that virtual memory and single-stepping are common to most if not all processor architectures and, a TSS (or an equivalent) is typically found on processors supporting legacy I/O such as the IA-32 (and compatible) processors. Figure 5.1 illustrates the architecture of VAMPiRE in its current version.

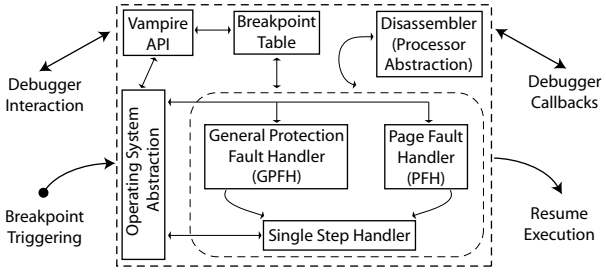


Figure 5.1. VAMPiRE Architecture.

The core of the framework is composed of a *page-fault handler* (PFH), a *general protection fault* handler (GPFH), a *single-step handler* and a *framework API*. The PFH provides breakpoints to be set on code, data and memory-mapped I/O while the GPFH provides legacy I/O breakpoint support. A debugger employing VAMPiRE interacts with

the framework through its API to set and/or remove breakpoints at desired locations. Every breakpoint has an associated *callback* (provided by the debugger), a function to which control is transferred upon breakpoint triggering.

When a breakpoint triggers, the framework fault handlers (PFH or GPFH) receive control, determine the type of breakpoint (by employing a *disassembler*) and invoke the callback to do the processing and resume execution. The single-step handler is employed by the framework for breakpoint persistence, a feature that allows a breakpoint to re-trigger automatically in the future. A breakpoint under VAMPiRE can be persistent (re-triggerable) or non-persistent (one-shot) and can be active or inactive at any instant. VAMPiRE uses a *breakpoint-table* — a memory structure specific to the framework — to maintain a list of breakpoints set using the framework. The information stored in the breakpoint-table is used to identify a breakpoint (and its callback) during breakpoint triggering and processing.

5.3 Stealth Breakpoints

Our goal in designing and implementing VAMPiRE was twofold. First, it should be able to provide unlimited breakpoints to be set on code, data and I/O with the same precision as hardware breakpoints. Second, the breakpoints should be undetectable and impossible to circumvent. The following sections describes how VAMPiRE achieves these capabilities.

5.3.1 Stealth Code, Data and Memory-mapped I/O Breakpoints

VAMPiRE uses the virtual memory system of the underlying platform to provide breakpoints on code, data and memory-mapped I/O. Support for virtual memory on most processors, is in the form of page directories and page tables. The memory addressing space is divided into chunks of equal size elements called a *page*. (typically 4K or 8K

bytes in size). Pages can have various attributes such as `read`, `read/write`, `present`, `not-present`, `user`, `supervisor` etc. These attributes along with a page-fault exception is used to provide virtual memory support and memory protection.

A page-fault exception is generated by the underlying processor when a reference to a page is inconsistent with the page attributes (e.g. a write is issued to a location in the referenced page, while the page attribute is `readonly`). The OS normally installs a handler for the page-fault exception, which implements a paging algorithm, provides protection and other features such as copy-on-write etc. VAMPiRE installs its PFH (pseudo-code shown in Figure 5.2), replacing and chaining to the existing handler to implement breakpoints on code, data and memory-mapped I/O.

```
 1. obtain linear-address of fault
 2. find memory page corresponding to linear-address
 3. if (no active breakpoints on memory page){
 4.   apply clock patch
 5.   chain to previous page-fault handler
 6. }

 7. mark memory page present preventing recursive faults
 8. find effective address of instruction causing the fault
 9. if (effective address is a breakpoint address){
10.   get breakpoint type (read, write and/or execute)
11.   trigger breakpoint and process
12. }

13. setup re-triggering for persistent breakpoints on
    memory page
14. setup single-step handler for breakpoint persistence
15. apply clock patch
16. return
```

Figure 5.2. VAMPiRE PFH.

To set a breakpoint (code, data or memory-mapped I/O) at the desired memory location, VAMPiRE sets the attribute of the page corresponding to the memory location to `not-present`. This results in a page-fault exception when any location is referenced in that page. When VAMPiRE's PFH gets control it obtains the linear-address of the fault (this is passed to the handler by the processor) and determines the page corresponding to the linear-address. The PFH then performs a check to determine if the page corresponding to the linear-address contains any active breakpoints. If not, the PFH applies a clock

patch (if applicable) to hide its processing latency from the malware (see Chapter 9) and chains to the original page-fault handler since the fault is not due to the framework. This is shown in lines 1–6, Figure 5.2.

If the page corresponding to the linear-address of the fault, contains active breakpoints, the PFH sets the page attribute to `present` to prevent recursive page-faults within the handler. A *disassembler* is then employed to obtain the effective address of the instruction causing the fault. If the effective address is a breakpoint address, the disassembled instruction is analysed to see if it is a read, write or execute breakpoint for code, data and memory-mapped I/O. If so, the breakpoint is triggered and processed. This is shown in lines 7–12, Figure 5.2.

The PFH then prepares to re-trigger any persistent breakpoints on the page corresponding to the linear-address causing the fault. This is accomplished by populating a global breakpoint re-trigger array (see Section 5.5). A single-step handler is then setup to step over the current instruction that caused the breakpoint to enable breakpoint persistence (see Section 5.4). At this point, the PFH applies a clock patch (if applicable) to hide the latency of the handler from the malware (see Chapter 9). Finally, the PFH returns, marking the end of exception. This is shown in lines 13–16, Figure 5.2.

5.3.2 Stealth Legacy I/O Breakpoints

Legacy I/O uses processor supported I/O instructions to read from or write to a I/O port. Legacy I/O breakpoints involve breakpoint triggering due to such I/O instructions attempting to read from or a write to a specific I/O port. Processors which support legacy I/O along with virtual memory, support legacy I/O protection or virtualization in the form of a TSS (or an equivalent). VAMPiRE relies on the TSS to provide breakpoints on legacy I/O on processors that support them. The TSS consists of a bitmap structure called the I/O Bitmap which is a bit array with 1 bit for every legacy I/O port in the

system. If the bit corresponding to a I/O port is set to a 1, the processor causes a general protection fault (GPF) when I/O instructions referencing that I/O port are executed. VAMPiRE makes use of this feature to support legacy I/O breakpoints. It installs its GPFH (pseudo-code shown in Figure 3), replacing and chaining to the existing handler to implement breakpoints on legacy I/O.

```
1. mask out effect of the trap flag

2. for(each entry in breakpoint re-trigger array){
3.   if (breakpoint type is legacy I/O)
4.     set corresponding bit in I/O bitmap to 1
5.   else
6.     set target memory page attribute to not-present
7. }

8. restore trap flag and uninstall handler
9. return
```

Figure 5.3. VAMPiRE GPFH.

VAMPiRE's GPFH semantics is very similar to that of its PFH. To set a breakpoint at the desired I/O location (read or write), VAMPIRE sets the bit corresponding to the I/O port, in the I/O Bitmap array to a 1. This results in a GPF when any access is attempted using that particular I/O port. When VAMPiRE's GPFH gets control, the handler obtains the linear address of the fault via the TSS and disassembles the instruction causing the fault. If the instruction does not belong to the category of legacy I/O instructions, the GPFH applies a clock patch (if applicable) to hide its processing latency from the malware (see Chapter 9) and chains to the original GPF handler since the fault is not due to the framework. This is shown in lines 1–6, Figure 5.3.

If the instruction causing the fault is identified as a legacy I/O instruction, the GPFH checks to see if the corresponding I/O port (extracted from the instruction) has a breakpoint set on it. If so, the corresponding I/O permission bit in the I/O Bitmap is set to a 0 to prevent recursive faults from accessing that port within the handler. The GPFH then obtains the breakpoint type (read or write) and processes the breakpoint.

This is shown in lines 7–12, Figure 5.3. The rest of the GPFH processing (lines 13–16, Figure 5.3) is the same as described for the framework PFH (lines 13–16, Figure 5.2, Section 5.3.1) for the framework PFH.

As with memory and legacy I/O breakpoints in user-mode, VAMPiRE supports legacy I/O breakpoints in kernel-mode, without making use of any processor supported debbugging aid. This feature is useful in situations involving analysis of malware employing legacy I/O accesses (e.g keyloggers). To support Legacy I/O breakpoints in kernel-mode, VAMPiRE uses the concept of stealth localized-executions (SLE, see Chapter 7). It establishes overlay and release points on one or more entry points in the host OS which lead to execution of kernel-mode code. When such entry points are reached by an executing target code stream, the framework starts SLE on the code-stream and sets up SLE to respond to events involving legacy I/O acess. The events generated by SLE are then handled by the framework to determine if a legacy I/O breakpoint has occured. Chapter 7 discusses SLE in detail.

The choice of the entry points depends upon the host OS, its support for kernel-mode code and the malware being analyzed, but a few general entry points are system calls, kernel-mode thread pre-emptions (scheduler), kernel-mode thread creation, kernel-mode exceptions etc. As an example we used the CreateProcess, RegisterService, PSCreateSystemThread, KiSwitchContext and the IRQ-1 handler as entry points during our analysis sessions involving the Troj/Feutel-S [65], a keylogger malware running on the Windows OS.

VAMPiRE can monitor a arbitrary range of code-stream in kernel-mode, a single or a group of entire kernel-mode threads, a kernel-mode exception handler or a system call invocation depending on the overlay/release points that are established. While, VAMPiRE can run all kernel-mode code employing localized-executions thereby able to trap any access to given legacy I/O ports, it is seldom used in this fashion since

running all the kernel-mode code can result in substantial overhead during the analysis process. Instead, the individual performing the analysis typically instructs the framework to deploy SLE on selected kernel-mode code streams to trap accesses to legacy I/O port access within the selected code stream.

As an example, let us consider Troj/Feutel-S. This keylogger uses legacy I/O access to the keyboard in order to capture user input. The malware performs legacy I/O access to the keyboard controller in kernel-mode by setting up its own IRQ-1 handler as well as in its own kernel-mode threads. Thus, an individual in this case might choose to set an overlay /release point as the IRQ-1 handler. Alternatively the individual could also use KiSwitchContext, an internal Windows OS function that does kernel-mode thread pre-emption, with a filter on the process ID of the malware to catch legacy I/O access in its kernel-mode threads.

5.4 Breakpoint Persistence

The single-step exception is a standard exception on all processor architectures that is triggered upon execution of each instruction when a certain processor flag (also called the trap flag) is activated. VAMPiRE installs its own single-step handler (pseudo-code shown in Figure 5.4), replacing the existing handler to implement breakpoint persistence. The single-step handler is installed on demand from the framework fault handlers (PFH and GPFH).

When the single-step handler is invoked due to a singlestep exception , the handler first makes sure that any effect of the trap flag (indicator for single-step) is masked out of the instruction that has just been stepped over. This is a very important step towards hiding the framework from the malware being analysed (see Chapter 9). The single-step handler then iterates through every element in the breakpoint re-trigger array (see Section 5.5) and sets the appropriate bit in the I/O Bitmap to 1 (in case of legacy I/O

```
 1. obtain linear-address of the fault via TSS
 2. disassemble the instruction causing the fault
 3. if (not legacy I/O instruction){
 4.   apply clock patch
 5.   chain to previous GPF handler
 6. }

 7. determine I/O port in disassembled instruction
 8. if (I/O breakpoint on I/O port){
 9.   reset I/O bitmask for the port
10.   find breakpoint type (read or write)
11.   trigger breakpoint and process
12. }

13. setup re-triggering for persistent breakpoints on
    memory page
14. setup single-step handler for breakpoint persistence
15. apply clock patch
16. return
```

Figure 5.4. VAMPiRE Single-Step Handler.

breakpoint) or sets the appropriate memory page attribute to `not-present` (in case of code, data or memory-mapped I/O breakpoint). This ensures that future accesses to the page or the legacy I/O port re-triggers the breakpoint, thereby achieving breakpoint persistence. Finally, the handler resets the trap flag, uninstalls itself, and issues an end of exception. This is shown in lines 1–9, Figure 5.4.

5.5 Breakpoint Table, Callbacks, Re-trigger Array

VAMPiRE makes use of certain important storage elements for its functioning. Chief among them is the *breakpoint-table*, which is an array of structures, one element for each breakpoint that is set using the framework. The breakpoint-table (shown in Figure 5.5) consists of (1) the breakpoint address (2) the breakpoint type (code, data, memory mapped I/O or legacy I/O), (3) the breakpoint attributes which include read, write, execute and the persistence flags, (4) the address of the callback which gets control when the breakpoint is triggered, and (5) the breakpoint status, which indicates if the breakpoint is currently active or inactive.

A *callback* is simply a function that is supplied with parameters identifying the breakpoint address, the breakpoint type (code, data, memory mapped or legacy i/o), the breakpoint condition (read, write or execute) and the breakpoint attribute (persistent

Breakpoint Address	Type	Attributes	Callback Address	Status
00501020h	Data	R	812E5000h	✓
10005000h	Code	R W X P	812E5000h	✓
50h	Legacy I/O	R	812E5000h	✗
50001200h	Code	X	812E5000h	✓

R = Read W = Write X = Execute P = Persistent ✓ = Active ✗ = Inactive

Figure 5.5. VAMPiRE Breakpoint Table.

or non-persistent). A debugger using VAMPiRE will have to provide such a function that gets control when a breakpoint is triggered to do the appropriate processing. The breakpoint address is the address in memory, which would trigger a breakpoint when accessed. The breakpoint address is the I/O port number if the breakpoint is of type legacy I/O. The breakpoint address and its attributes are usually set by the debugger using the appropriate VAMPiRE API. Though a breakpoint can be set as persistent or non-persistent during its creation, a callback can subsequently over-ride this at runtime by returning an appropriate integer value in the breakpoint attribute parameter, to determine if the breakpoint is to remain persistent or a one shot. One could use different callbacks for different breakpoints, or use a single callback to handle all the breakpoints set using the framework.

Figure 5.5 shows several examples of entries one may find in the breakpoint table. The first entry shows a breakpoint set on memory read for the address 00501020h. The breakpoint will only be triggered once since the persistent attribute is missing. Similarly, the second entry shows a breakpoint being set on memory read, write, and/or execute at location 10005000h with the breakpoint being re-triggered automatically because of the presence of the persistent attribute. The third entry shows a legacy I/O breakpoint being set for a read on port 50h that is currently inactive. Also from Figure 5.5 one can see that all breakpoints are routed through one single callback function at address 812E5000h.

VAMPiRE uses an array, called the *breakpoint re-trigger array* in its fault handlers. Each entry in the breakpoint retrigger array is a structure containing the breakpoint type, the target page of the breakpoint address (for code, data and memory-mapped I/O) or the target I/O port number (in case of legacy I/O), and its associated callback address. This array is used by the single-step handler (see Section 5.4) to enable breakpoint persistence.

5.6 Performance

The performance of a breakpoint framework such as VAMPiRE depends on a number of factors, chief among them being the nature of the code being debugged, the dependency of the code on the data accessed, and the style of debugging employed by an individual in terms of setting breakpoints. These factors are not easy to characterize and hence it is difficult to come up with a representative analysis session for performance measurements. This is further complicated by the fact that the same individual can adopt a different style of analysis at any given time for a given code fragment. Therefore, we will concentrate on presenting the performance of VAMPiRE based on analysis sessions with a Windows based virus, `W32.HIV` [46]. The performance of the framework for other analysis sessions can be estimated in a similar fashion.

To validate VAMPiRE, we used WiLDCAT with our prototype debugger code-named ViPER. The current version of our debugger makes use of VAMPiRE and also has support for traditional hardware and software breakpoints. For test purposes, an AMD Athlon XP 1.8 GHz processor with 512 MB of memory was used. Readings were taken at various points within our debugger after a breakpoint was set and before the triggering of a breakpoint and its processing. We used processor clock cycles as the performance metric for comparison. This metric is chosen, as it does not vary across processor speeds and also since it is a standard in literature related to micro benchmarks. The `RDTSC` instruction was used to measure the clock cycles.

The first part of this section will present the performance measurements of VAMPiRE when compared to traditional hardware breakpoints. This is from our analysis session involving code fragments of the W32/HIV where both VAMPiRE and hardware breakpoints can be used. The second part of this section will then present the performance measurements of VAMPiRE for code fragments where hardware breakpoints cannot be used thereby showing the utility of the framework. For the graphs in this section, the x-axis is the amount of clock cycles that have elapsed between executing the code with the breakpoint set and breakpoint triggering. The y-axis (category axis) represents the code fragments which were chosen arbitrarily from our debugging sessions with the `W32.HIV`. Also, a part of the graph is magnified (indicated by dotted lines) to provide a clear representation of categories with low values on the x-axis.

The performance measurements comparing hardware breakpoints and VAMPiRE are shown in Figure 5.6. With VAMPiRE, for purposes of measurement, code being debugged and breakpoints fall under two categories. The first is when the code being debugged and the breakpoints are on separate memory pages. The second is when the code and the breakpoints are on the same page. The latency due to our framework for both cases with various self-modifying and self-checking virus code fragments is shown in Figure 5.6.

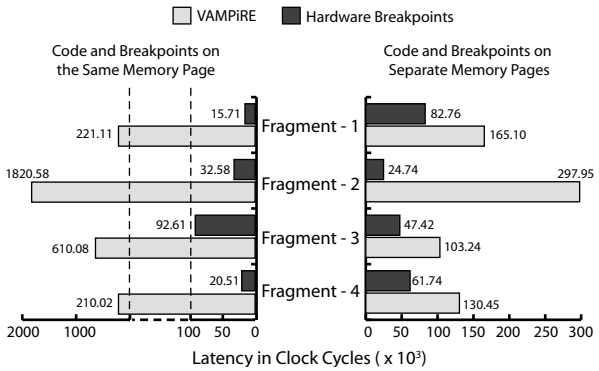


Figure 5.6. Performance of VAMPiRE vs Hardware Breakpoints.

From the graph we can see that the performance of VAMPiRE is comparable to hardware breakpoints in the case where code and breakpoints exist on different memory pages. However, in some cases (as exemplified by Fragment-2, Figure 5.6, for code and breakpoints on different memory pages) when the data and/or code reference to a page is much higher, we find that the latency of VAMPiRE is much more than that of hardware breakpoints. When we look at code and breakpoints existing on the same memory page, the latency of VAMPiRE is inherently higher due to its design and the single-step mechanism coming into effect (see Section 5.4).

Figure 5.7 shows the performance measurements for VAMPiRE for the code fragments of the W32/HIV where traditional breakpoints could not be used. A point to be noted is that the graph is not a comparison (since hardware breakpoints cannot be used to analyse such code fragments) but is only provided for the sake of completeness of performance evaluation.

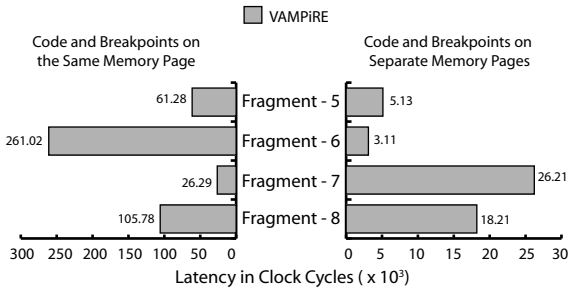


Figure 5.7. Performance of VAMPiRE.

Thus we can conclude that the performance of the framework is highly dependent on the nature of code being debugged and the nature of the breakpoints set by an individual. However, the measurements give a gross indication of the latency one might expect. As seen from Figure 5.6 and Figure 5.7, the latency of the framework is found to be well within the limits to suit interactive debugging.

5.7 Summary

The features provided by standard aids for stalling code execution during runtime (hardware and software breakpoints) do not suffice to study code employing SM-SC and/or any form of anti-analysis. With VAMPiRE however, this task is greatly simplified. The framework allows unlimited number of breakpoints to be set on code, data, and I/O with the same precision as that of hardware breakpoints. The breakpoints set by VAMPiRE cannot be detected or countered and the latency of the framework is well suited for interactive debugging as seen from its performance measurements.

CHAPTER 6

FINE-GRAINED MALWARE ANALYSIS

In this chapter we discuss the design implementation and performance of Cobra, the subsystem of WiLDCAT, that incorporates radical techniques empowering the environment to perform fine-grained code-analysis at runtime on malware code-streams.

6.1 Introduction

Fine-grained malware analysis is a challenging task that provides important pieces of information that are key to building a blueprint of the malware core structure and functioning, that aids in the detecting and countering the malware and its variants. As an example, the W32/MyDoom trojan with variants commonly known as W32/MyDoom.X-MM (where X can be A, B, R, S, G etc.) share the same metamorphic code layers, encryption/decryption engine and similar anti-analysis schemes. The Netsky [19], Beagle [86] and Sobig [69] worms are some other examples of how malware are coded in an iterative fashion to add more features while retaining their core structure. Thus, once the core structure of a malware is documented, it becomes easy to tackle the rest of its variants as well as other malware which share a similar structure. Also, with malware writers employing more complex and hard to analyze techniques, there is need to perform fine-grained analysis of malicious code to counter them effectively.

Current research in fine-grained malware analysis can be broadly categorized into static and dynamic approaches. Static approaches allow exhaustive fine-grained analysis because they are not bound to a specific execution instance. They allow detection of malicious code without actually running the program, ensuring that the malices discov-

ered will never be executed and incur no runtime overhead. In spite of such powerful properties, static analysis has some limitations. With static analysis there is the problem that the analyzed code need not be the one that is actually run; some changes could be made between analysis and execution. This is particularly true with polymorphism [66, 7] and metamorphism [73] that are techniques employed by most if not all next generation malware. Also it is impossible to statically analyze certain situations due to undecidability (eg. indirect branches). Further, static code analysis also has limitations related to code obfuscation, a technique used by malware to prevent their analysis and detection.

Dynamic approaches overcome these limitations by analyzing the code during runtime, ensuring that the the analyzed code is the one that is actually run without any further alterations. Though there have been several research on dynamic coarse-grained malware analysis [5, 27, 34, 84, 38], not much has been published about dynamic fine-grained malware analysis. Currently dynamic fine-grained malware analysis can be achieved by employing debuggers and/or fine-grained instrumentation frameworks. When using a debugger such as Softice [53], WinDBG [82] etc., the basic approach is to set breakpoints on identified areas and then trace the desired code-stream one instruction at a time to glean further information. Alternatively one could also employ a fine-grained instrumentation framework such as Pin [41], DynamoRIO [36] etc., for automated tracing of code-streams. However, these tools are not equipped to handle malicious code and have severe shortcomings in the context of malware.

Current debugging and fine-grained instrumentation techniques can be easily detected and countered by the executing malware code-streams. As an example, W32/HIV [46], W32/MyDoom [44], W32/Ratos [74], and their variants employ techniques such as code execution timing, where the malware time their executing code thereby easily detecting that they are being analyzed (since debugging and/or automated code tracing incur la-

tency that is absent during normal execution). Further they contain ad-hoc detection schemes against popular debuggers such as Softice, WinDBG, etc. Current debugging and fine-grained instrumentation techniques do not carry support for self-modifying and/or self-checking (SM-SC) code. Most if not all malware are sensitive to code modification, employing subtle anti-analysis techniques and code obfuscations that defeat breakpoints in debugging and the process of automated code tracing using fine-grained instrumentation frameworks. For example, W32/MyDoom and W32/Ratos employ integrity checking of their code-streams with program counter relative code modification schemes which render software breakpoints and current fine-grained instrumentation frameworks unusable.

Malware that execute in kernel-mode are even tougher to analyze using current dynamic fine-grained techniques, since they have no barriers in terms of what they can access. For example, W32/Ratos employs a multithreaded polymorphic/metamorphic code engine running in kernel-mode, and overwrites the interrupt descriptor table (IDT) with values pointing to its own handlers. Current fine-grained instrumentation frameworks do not handle kernel-mode code and do not carry adequate support for multithreading. Current debugging techniques provide kernel-mode code support but do not support multithreading in kernel-mode. Furthermore, recent trend in malware has been to employ debugging mechanisms supported by the underlying processor within their own code, thereby effectively preventing analysis of their code using current debugging techniques. Examples include W32/Ratos, which employs the single-step handler (used for code tracing) to handle its decryption in kernel-mode and W32/HIV which uses debug registers (used for hardware breakpoints) for its internal computation.

Cobra overcomes the shortcomings of existing strategies related to fine-grained code-analysis by using the concept of stealth localized-executions to enable dynamic fine-grained malware analysis in commodity OSs in a completely stealth fashion. It provides a stealth, efficient, portable and easy-to-use framework that supports multithreading, SM-

SC code and code obfuscations in both user- and kernel-mode while allowing selective isolation of malware code-streams. By *stealth* we mean that Cobra does not make any visible changes to the executing code and hence cannot be detected or countered. The framework employs subtle techniques such as block-coalescing and skipping to provide an *efficient* supervised execution environment. The framework supports what we call *selective-isolation*, that allows fine-grained analysis to be deployed on malware specific code-streams while allowing normal code-streams to execute as is. The framework also allows a user to tie specific actions to events that are generated during the analysis process in real-time.

6.2 Overview of Cobra

Our goal in designing and implementing Cobra was twofold. First, it should be able to provide a stealth supervised environment for fine-grained analysis of executing malware code-streams, supporting multithreading, self-modifying code and any form of code obfuscation in both user- and kernel-mode on commodity OSs. Second, one must be able to deploy the framework dynamically and selectively on malware specific code-streams while allowing other code-streams to execute as is.

Fine-grained malware analysis using Cobra is facilitated by a technique that we call *stealth localized-executions*. The basic idea involves decomposing a code-stream into several groups of instructions — which we call *blocks* — that are then executed, one at a time, in a fashion so as to mimic the normal execution of the target code-stream. Each block is implanted with various invisible Cobra specific code constructs (as applicable), ensuring that the framework has complete control over the executing code-stream while remaining stealthy.

Figure 6.1 illustrates the current architecture of Cobra. The framework core consists of a Block Create and eXecute Engine (BCXE), a disassembler, a block repository, a

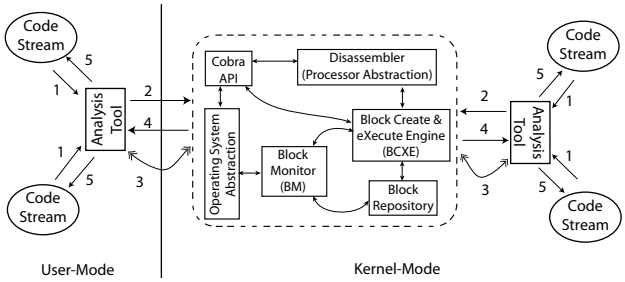


Figure 6.1. Architecture of Cobra.

block-monitor, and a framework API. The BCXE is responsible for decomposing a target code-stream into blocks and executing them. The target code-stream can be a regular or a malware specific code-stream. An architecture specific *disassembler* is employed to construct blocks corresponding to the target code-stream, in a dynamic fashion. The *block repository* functions as a local cache for storage of blocks. Only code from the block repository is executed and never any code from the target code-stream. Localized-executions on a target code-stream begin at an *overlay point* and stop at a *release point*, a user-defined range where fine-grained analysis is desired. This allows Cobra to be deployed and removed in a dynamic fashion on a given code-stream while allowing other code-streams to execute as is, a technique we call *selective isolation*. As an example, the W32/Ratos trojan and its variants employ several kernel-mode threads for their functioning. An overlay point in this case could be KiSwitchContext (a Windows OS internal kernel function responsible for thread pre-emption) and a release point could be the return from this function. The *block-monitor* employs subtle techniques involving virtual memory to protect critical memory regions during localized-executions and is responsible for maintaining coherence between the target code-stream and its blocks in case of self-modifying code.

As Figure 6.1 shows, there are typically three binary elements present during an analysis session: the target code-streams (residing in either user- and/or kernel-mode),

the analysis tool employing Cobra (typically some sort of debugger), and Cobra itself. The analysis tool, for each overlay point in a target code-stream, invokes the framework for fine-grained analysis over a specified range of the code-stream (Steps 1 and 2, Figure 6.1). The analysis tool then performs the required actions (processing) for specified events during the execution of the blocks. An event can be: block creation, execution of instructions within a block, acesss to critical memory regions within a block etc. (Step 3, Figure 6.1). Cobra stops localized-executions at the specified release point and transfers control back to the analysis tool which then resumes normal execution of the target code-stream (Steps 4 and 5, Figure 6.1).

Cobra resides in kernel-mode and can capture multithreaded, SM-SC code and any form of code obfuscation in both user- and kernel-mode with ease. The framework is completely re-entrant, as it does not make use of any OS specific routines during localized-executions and uses shared memory with its own isolation primitives for interprocess communication. The framework employs techniques such as *skipping* (where standard and/or non-standard code-streams are excluded from localized-executions) and *block-coalescing* (where multiple blocks are composed together) for efficiency.

6.3 Stealth Localized-executions

Cobra decomposes a target code-stream into several groups of instructions and executes them in a fashion so as to mimic the code-stream's normal execution. This process is what we call *localized-executions* and the instruction groups are called *blocks*. A block is nothing but a straight-line sequence of instructions that terminates in either of these conditions: (1) an unconditional control transfer instruction (CTI), (2) a conditional CTI, or (3) a specified number of non-CTIs. A *block-repository* contains a subset of the recently constructed blocks and acts as a framework local cache. Only blocks residing in the block-repository are executed — never the instructions in the target code-stream

(hence the term *localized-executions*). Cobra's BCXE is responsible for creating blocks from the target code-stream and executing them.

6.3.1 Block Creations

The BCXE employs an architecture specific disassembler on the target code-stream, to discover instructions one at a time, and create the corresponding blocks. Figure 2a shows part of a code-stream of the W32/Ratos trojan and a typical block creation process. The code-fragment has been modified to remove details not pertinent to our discussion and the instructions are shown in the 32-bit assembly language syntax of the IA-32 (and compatible) processors [31].

Every block ends with a framework specific set of instructions — which we call a *Xfer-stub* — that transfers control to the BCXE. Xfer-stubs ensure that Cobra is always under control of the target code-stream being analyzed. When a block is executed, the BCXE gets control at the end of the block execution via the block xfer-stub, determines the target memory-address to create the next block from, dynamically generates a new block for the corresponding code-stream if it has not done before, and resumes execution at the newly generated block. Thus, execution of blocks follows a path which is the same as the normal execution of the target code-stream in absence of the framework. Figure 6.2b shows the blocks created by the BCXE for the code-stream shown in Figure 6.2a.

The BCXE differs from VMs employed in current hypervisors [81, 4] and fine-grained instrumentation frameworks [41, 52, 11, 63, 43] in that: (a) it employs special treatment for CTIs thereby supporting SM-SC code and any form of code obfuscation (see Chapter 9), (b) it employs special treatment on privileged instructions and instructions that betray the real state of a code-stream and hence cannot be detected or countered in any fashion (see Chapter 9), (c) it achieves efficiency without recompiling the instruc-

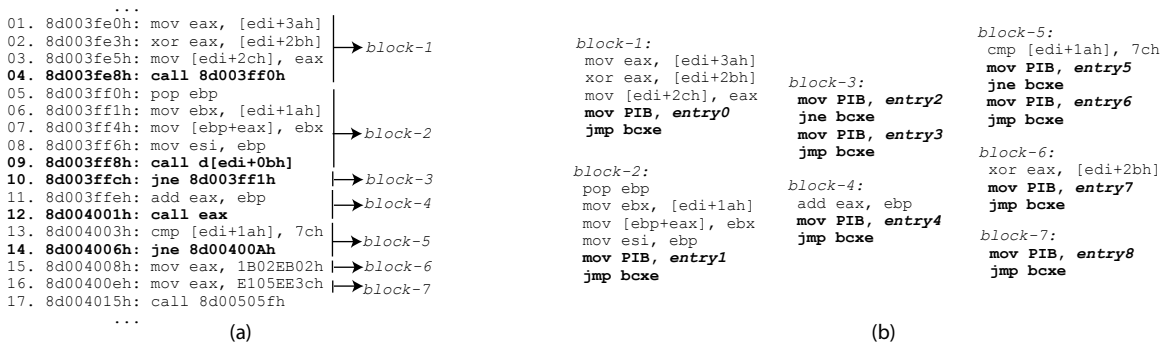


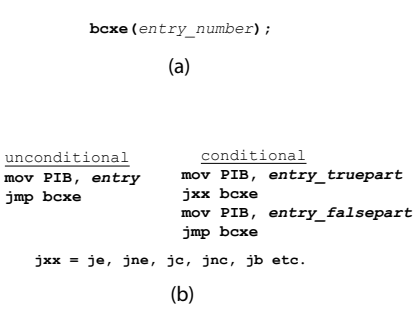
Figure 6.2. Block Creation: (a) Target Code-stream, and (b) Corresponding Blocks.

tions of the target code-stream (see Section 6.3.6), and (d) it is completely re-entrant supporting multithreading under both user- and kernel-mode and allows tuning the level of fine-grained analysis desired.

6.3.2 Xfer-Stubs

Xfer-stubs, Cobra specific code constructs that terminate every block, can be abstracted as a function that takes a single parameter (Figure 6.3a). The parameter is an index into a *Xfer-table*, an internal data structure of the framework, which enables the BCXE to obtain runtime information on the supervised code-streams, which include among other things, the address of the target code-stream to generate a new block from. A xfer-stub replaces the CTI (conditional or the unconditional) that terminates a block. In some cases, where block creation terminates because a predefined number of non-CTIs were reached, Cobra treats the block as ending with an unconditional branch/jump instruction and creates a corresponding xfer-stub. Figure 6.3b shows the xfer-stub implementations for conditional and unconditional CTIs on the IA-32 (and compatible) processors. For unconditional CTIs the corresponding xfer-stub simply performs an unconditional jump (JMP) into the BCXE. For conditional CTIs, the xfer-stub translates

a conditional into a conditional and an explicit JMP. This ensures that the BCXE gets control for both situations where the conditional evaluates to true and false. The parameter to a xfer-stub is passed via a Parameter Information Block (PIB) — a per-thread, framework internal memory area, instead of the thread stack. This is required to prevent Cobra from being detected or countered by the malware being analyzed (see Section 6.4).



Entry Number	Target Address Type (TAT)	Target Address Value (TAV)	Xfer-Stub Type	Xfer-Stub Parameters
entry0	VALIMM	8d003ff0h	XSPECIAL	8d003ff0h
entry1	VALIND	[edi+0bh]	XSPECIAL	8d003ffch
entry2	VALIMM	8d003ff1h	XNORMAL	NULL
entry3	VALIMM	8d003ffeh	XNORMAL	NULL
entry4	VALIND	eax	XSPECIAL	8d004003h
entry5	VALIMM	8d00400ah	XNORMAL	NULL
entry6	VALIMM	8d004008h	XNORMAL	NULL
entry7	VALIMM	8d004010h	XNORMAL	NULL
entry8	VALIND	[edi+eax*4]	XNORMAL	NULL

(c)

Figure 6.3. (a) Xfer-Stub Abstraction, (b) Xfer-Stub Implementation on IA-32 (and compatible) processors, and (c) Xfer-Table.

The xfer-table (shown in Figure 6.3c) is an array of structures, one element for each xfer-stub that is currently used by the framework. Every entry in the xfer-table consists of (1) a target-address type (TAT), (2) a target-address value (TAV), (3) the xfer-stub type (XST), and (4) additional xfer-stub parameters (if applicable). The TAT determines if the address at which the BCXE will create a new block from, is an immediate value (VALIMM) or an indirect expression (VALIND) whose value has to be evaluated by the BCXE at runtime upon entry from the xfer-stub. The TAV is a constant (when TAT is VALIMM) or an expression (when TAT is VALIND). The XST indicates whether the xfer-stub is for a standard CTI with no additional processing (XNORMAL) or a CTI that needs special processing (XSPECIAL). XST XNORMAL is used in the majority of cases while XST XSPECIAL is used to handle cases where: (1) the CTI uses the thread stack implicitly (CALL, INT etc.) (see Chapter 9), and (2) the framework needs

to employ block specific xfer-stubs to remain stealth (see Section 6.3.5). In both cases the framework makes use of additional xfer-stub specific parameters.

Figure 6.3c shows the entries corresponding to the xfer-stubs for the blocks shown in Figure 6.2b. As seen, entry0 has TAT set to VALIMM and TAV set to the constant 8d003ff0h since the corresponding xfer-stub is for a CALL instruction (Line 4, Figure 6.2a) which deals with a constant target address at which the BCXE generates the next block from. However, entry4 has TAT set to VALIND and TAV set to the expression EAX since the corresponding xfer-stub is for a CALL instruction (Line 12, Figure 6.2a) whose target address depends on the runtime value of EAX at that point in execution. For both cases, the XST is set to XSPECIAL to indicate that the xfer-stubs require additional processing.

6.3.3 Block Executions

Localized-executions start from a user-defined point — which we call an *overlay point* — in a target code-stream. An overlay point is the memory-address (typically a OS and/or a library function address) in a target code-stream from where fine-grained analysis is desired. An overlay point under Cobra is defined by employing VAMPiRE. Once execution reaches an overlay point, Cobra is invoked to start fine-grained analysis until a *release point* is reached. A release point is the memory-address in a target code-stream where Cobra relinquishes supervision and lets the code-stream execute in a normal fashion. A overlay point and its corresponding release point thus establish a fine-grained analysis range on a target code-stream under Cobra, while allowing other code-streams to execute as is — a technique we call *selective isolation*. Under Cobra, one can specify multiple overlapping and/or non-overlapping overlay and release points for a target code-stream. The framework also supports nesting of overlay and release points and allows release points to be infinite, in which case the complete thread containing the target

code-stream is constantly run under Cobra's supervision, until the thread terminates or the framework is invoked to stop localized-executions.

As an example, the W32/Ratos trojan runs under the Windows OS and employs several kernel-mode threads for its inner functioning. One such kernel-mode thread replaces the default single-step handler in the Interrupt Descriptor Table (IDT) with a trojan specific handler. With Cobra, we employ *KiSwitchContext* (an internal Windows kernel function responsible for thread pre-emption) as the first overlay point to execute each of the trojan kernel-mode threads under the supervision of Cobra with infinite release points. Upon detection of an access to the single-step vector in the IDT via a Cobra generated event (see Section 6.3.4), we employ the destination address of the single-step handler as our second overlay point (with the corresponding release point being the return from exception), thereby allowing us to study the W32/Ratos single-step handler in further detail. All this is done while co-existing with other OS user- and kernel-mode threads and exception handlers.

Cobra's BCXE executes individual blocks in an unprivileged mode regardless of the execution privilege of the target code-stream. This ensures that Cobra has complete control over the executing instructions. The framework can also monitor any access to specified memory regions, the OS kernel and resources, dynamic libraries etc. Cobra employs the virtual memory system combined with subtle techniques for memory access monitoring. On the IA-32 (and compatible) processors, for example, Cobra elevates the privilege level of specified memory regions and critical memory structures such as page-directories/page-tables, the IDT, the descriptor tables (GDT and LDT), task state segments (TSS) etc. by changing their memory page attributes and installs its own page-fault handler (PFH) to tackle issues involving memory accesses. The PFH also facilitates hiding framework specific details in the page-table/page-directories and the IDT while at the same time allowing a code-stream to install their own handlers and descriptors in

these tables. Cobra employs stealth-implants (see Section 6.3.5) to support supervised execution of privileged instructions in the event that the target code-stream runs in kernel-mode.

Cobra does not make use of any OS specific functions within its BCXE. The disassembler employed by the framework is completely re-entrant. The framework employs a per-thread PIB for the block xfer-stubs, does not tamper with the executing stack and employs subtle techniques to remain stealth (see Chapter 9). These features enable the framework to support multithreading since the executing threads see no difference with or without Cobra in terms of their registers, stack contents or time block. Cobra also supports automatic thread monitoring for a specified process or the OS kernel. This is a feature that automatically blocks every code-stream associated with a target process. Thus an entire process can be executed under Cobra by specifying the process creation API as an intial overlay point and allowing the framework to automatically insert overlay points thereafter on every new thread or process associated with the parent.

6.3.4 Events and Callbacks

Cobra generates various events during block execution. These include block creations, stealth implants, begin/end execution of a whole block, execution of individual and/or specific instructions within a block, system calls and standard function invocations, access to user defined memory regions, access to critical structures such as page-directories/page-tables, IDT, GDT etc. An analysis tool employing Cobra can employ event specific processing by registering *callbacks* — functions to which control is transferred by the framework to process a desired event during block execution. Callbacks are passed all the information they need to change the target code-stream registers and memory (including the stack), examine the block causing the event and instructions within it. A callback can also establish a new overlay point during an analysis session.

Events and Callbacks thus facilitate tuning the level of fine grained analysis from as fine as instruction level to progressively less finer levels.

A typical analysis process in our experience would employ events on block creations, begin/end of block executions, access to critical memory regions and any stealth implants before doing an instruction level analysis of blocks. As an example, if one considers the W32/Ratos, it overwrites the IDT single-step handler entry with a value pointing to its own single-step handler within a polymorphic code envelope. A IDT-write event can be used to obtain the trojan single-step handler address in the first place. The callback for the IDT-write event would use the single-step handler address as an overlay point to further analyze the trojan single-step handler in a fine-grained fashion. The events of block creation, begin/end block executions can then be used to build an execution model of the polymorphic code envelope. The inner working of the single-step handler can then be studied by an instruction level analysis on identified blocks.

6.3.5 Stealth Implants

Cobra scans a block for privileged instructions and instructions that betray the real state of the executing code-stream and replaces them with what we call *stealth-implants*. These are Cobra code constructs that aid in supervised execution of privileged instructions and the framework stealthness, while preserving the semantics of the original instructions in the target code-stream. Stealth-implants only take place on blocks and never on the original code. Thus they are undetectable by any intergrity checks, as such checks always operate on the original code-stream. Cobra inserts stealth-implants for various instructions and employs a host of antidotes for various possible ways in which a malware could detect the framework. Figure 6.4 shows stealth-implants for various instructions on the IA-32. Chapter 9 discusses techniques that can be used to detect the Cobra and how stealth implants help in countering them.

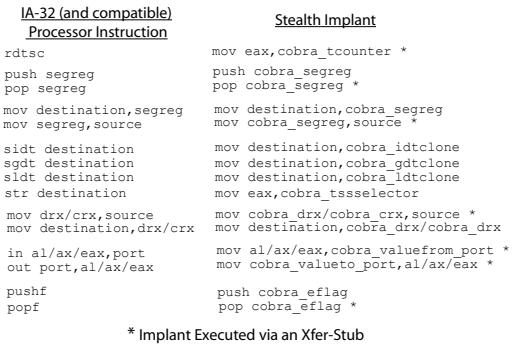


Figure 6.4. Stealth Implants.

Cobra replaces privileged instructions with xfer-stubs that transfer control to the BCXE which then locally executes these instructions in a controlled manner and returns the result. For example, Cobra replaces the RDTSC instruction with a regular MOV instruction that stores the value of Cobra's internal processor-counter to the EAX register. A point to note is that not all stealth-implants transfer control to the BCXE. Most instructions which store a value into a destination memory operand can have a stealth-implant without an xfer-stub directly replacing the instruction.

6.3.6 Block Skipping and Coalescing

A block may contain an instruction that transfers control to a code-stream following a standard execution semantic (e.g system calls, standard library calls etc.). Localized-executions of such standard code-streams result in increased latency and are in most cases undesirable (since they do not form a part of the malware being analyzed). For example, consider the code fragment of the W32/Ratos trojan as shown in Figure 6.5. The CALL instruction in line 8, Figure 6.5 uses an indirect target address, but is found to transfer control to a standard code-stream (in this case the VirtualProtect system call). Localized-execution of the system call in this case is meaningless since it does not contribute towards the analysis process. Note that the system call invocation does have a

bearing on the malware functionality at a coarse level, but does not have any implication on the fine-grained analysis of the malware code-stream. Cobra can identify such standard code-streams dynamically and exclude them from localized-executions, thereby reducing the latency that might occur in trying to execute an OS or standard code-stream. We call this technique *skipping*. Skipping can also be applied to non-standard code-streams during the analysis process. This might be used to exclude already analyzed malware specific code-streams. As an example, line 5 of Figure 6.5 shows a CALL instruction which performs an integrity check over a specified region of code. The code-stream concealed behind this CALL never changes in its semantics and can be thus be skipped after analyzing it once.

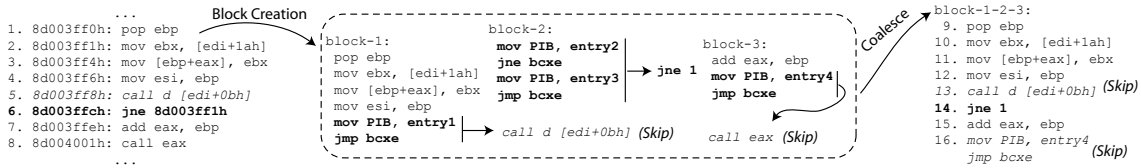


Figure 6.5. Skipping and Block-Coalescing.

Execution of blocks under Cobra involve control transfers to and from the BCXE. These transfers result in the saving and restoration of the processor registers which contribute to the framework latency. This is more important for code-streams employing loops since every iteration of the loop will involve invoking the BCXE. The framework employs a technique that we call *block-coalescing* to minimize latency due to such code constructs. In this technique, a group of blocks are brought together to form a single block, preventing multiple control transfers to the BCXE. Let us consider the code fragment shown in Figure 6.5. Here we see a loop implementing some form of integrity check. Figure 6.5 also shows the blocks associated with the code fragment. From our analysis session, we found the loop to execute close to 50 times on an average. Figure 6.5

shows the block-coalesced version, where blocks 1–3 have been coalesced, reducing the number of transfers to the BCXE. Block-coalescing is a powerful mechanism that can generate blocks which are very similar to the original code-stream, and can be executed with minimal latency while ensuring that Cobra is still under control of the executing code-stream. Block-coalescing is performed by a user-defined callback that chooses the blocks to participate in the coalescing process. With the above example, the first few instance of the loop executes normally producing blocks 1–3, Figure 6.5. Once the instructions in the blocks are analyzed the blocks can be coalesced together so that future iterations execute with minimal latency. Note from line 16, Figure 6.5 that the BCXE gets control once again after the loop has been executed without any intervention.

Cobra supports block-coalescing for self-modifying code by employing a subtle technique involving virtual memory. The idea is to set the memory page attributes of the executing code-streams to read-only. Thus, when a write occurs to such code regions due to self-modification, Cobra's block-monitor gets control and the blocks corresponding to such code regions are purged and re-created subsequently — a process we call *block-purging*.

6.4 Cloning

A malware can access critical system structures in order to detect that it is being analyzed using Cobra. For example a malware might try to obtain the PFH address and compare it with the system default values (which for certain OSes lies within a fixed range regardless of their version) in order to detect the framework. Similarly it might try to check the page-attributes of certain memory regions (eg. its code and data) which can be different due to memory access monitoring by Cobra. Further a malware can also install its own fault handlers in the IDT for its functioning. Cobra uses a technique that we call *cloning* to hide the framework while at the same time allowing the malware

to access such critical structures. The framework maintains a copy of critical memory regions such as the page-tables/page-directories, IDT, GDT etc. that reflect their system default contents initially. The framework block-monitor tackles issues such as reads and/or writes to such critical structures by presenting the clone of these memory regions, thereby fooling the malware into thinking that it is operating on the original memory regions. Stealth-Implants for certain instructions involving control registers such as CR3 (used to obtain the page-directory base address) and instructions such as SIDT, SLDT and SGDT present the addresses of the cloned memory regions instead of the original.

6.5 Performance

The performance of a fine-grained malware analysis framework such as Cobra depends on a number of factors, chief among them being: (a) the nature of the code being analyzed such as SM-SC code, obfuscated code, etc. and (b) the style of analysis employed by an individual such as selecting the code-streams to be analyzed, the analysis ranges, the blocks to be coalesced, the code-streams to be skipped etc. These factors are not easy to characterize and hence it is difficult to come up with a representative analysis session for performance measurements. This is further complicated by the fact that the same individual can adopt different styles of analysis at different times for a given code-stream. Therefore, we will concentrate on presenting the performance of Cobra based on analysis sessions with W32/Ratos, a malware based on the Windows OSs (see Chapter 10). The performance of the framework for other analysis sessions can be estimated in a similar fashion. To validate Cobra, we make use of WiLDCAT under the Windows XP OS, on a 32-bit Intel 1.7 GHz processor with 512 MB of memory.

We divide the total runtime latency of Cobra into: (a) latency due to block creations, (b) latency due to xfer-stubs, (c) latency due to block purging, and (d) latency due to stealth-implants. Readings were taken at various points within WiLDCAT and

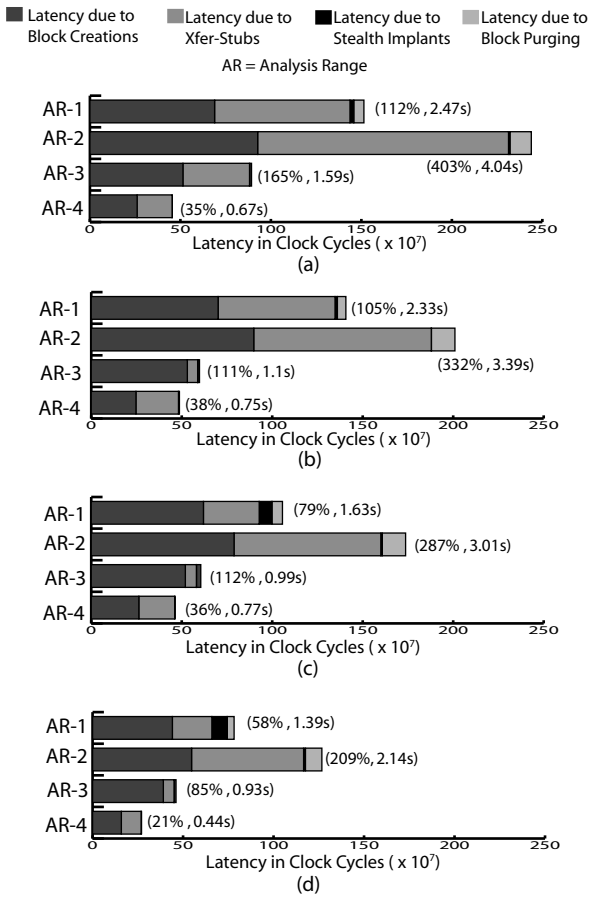


Figure 6.6. Performance of Cobra: (a) Normal, (b) With Block-Coalescing, (c) With Block-Coalescing and Skipping (standard), and (d) With Block-Coalescing and Skipping (standard and non-standard).

Cobra to measure these overheads. We use processor clock cycles as the performance metric for runtime latency. This metric is chosen, as it does not vary across processor speeds and also since it is a standard in literature related to micro benchmarks. The processor performance counter registers are employed to measure the clock cycles, using the RDMSR instruction.

Figure 6.6 shows the performance of Cobra employing different analysis methods for various analysis ranges. The analysis ranges were chosen from our analysis sessions involving the encryption and decryption engine of W32/Ratos. The choice of the analysis

ranges (single-step handler and parts of first and second decryption layers) were such that their semantics are relatively constant and they occur for every instance of the malware deployment. This allows us to obtain a deterministic performance measure of various aspects of the framework. For the graphs in Figure 6.6, the y-axis represents the analysis ranges and the x-axis is the amount of extra clock cycles that are incurred as opposed to the native runtime of that particular range. The data labels next to every stacked bar, in all the graphs, represent the percentage of normalized latency and its actual time in seconds for a 1.7GHz Intel processor.

Figure 6.6a shows the performance of Cobra when run normally (without applying performance enhancement techniques such as block-coalescing and/or skipping). As seen from the graph of Figure 6.6a, latency due to block-creations and xfer-stubs are present in every analysis range (analysis ranges 1–4 in this case) and form a major portion of Cobra's overall latency since these elements form the backbone of the framework. Latency due to block-purging only comes into effect when an analysis range involves self-modifying code (analysis ranges 1–3 in this case) and is due to the fact that the framework invalidates the blocks corresponding to the modified code regions. Latency due to stealth-implants occur when Cobra needs to patch a block in order to prevent its detection. This is shown in analysis ranges 1, 2 and 4 which contain W32/Ratos anti-analysis code fragments. In general, Cobra incurs lower overall latency when the ratio of straight line instructions to branches and loops is greater over a localized code region as exemplified by analysis range 4. In other cases, the overall latency of the framework depends upon the number and nature of the branches encountered in the code stream. As an example, analysis range 2 incurs a latency as high as 5 times its normal execution time. This is due to the presence of a high amount of code obfuscation via jumps, resulting in increased block creations and xfer-stub overheads for such blocks, for the analysis range.

Figure 6.6b shows the latency of Cobra employing block-coalescing on the same analysis ranges. Block-coalescing helps in reducing the latency due to xfer-stubs when analyzing code-streams involving loops. As seen from the graph in Figure 6.6b, analysis ranges 2 and 3 which contain a large number of loops incur a much lower overall latency with block-coalescing when compared to their overall latency without block-coalescing (Figure 6.6a). However, for analysis range 1 there is negligible gain in performance with block-coalescing, since it contains a very few number of loops. A point to note is that, the latency of analysis range 4 with block-coalescing is more than its latency without block-coalescing (Figure 6.6a). This is due to the fact that W32/Ratos, being metamorphic in nature, generates varying amount of code for a given functionality and embeds a random amount of anti-analysis code fragments in different instances of its deployment.

Skipping helps in further reduction of the overall latency by excluding standard and/or already analyzed code-streams from the analysis process. Figure 10c shows the performance of Cobra with block-coalescing and skipping applied to standard kernel code-streams. As seen from the graph of Figure 6.6c, the latency of analysis ranges 1 and 2 are further reduced as compared to their latency without skipping (Figure 6.6b). This is because, the code-streams in analysis ranges 1 and 2 invoke standard kernel functions such as VirtualProtect, KeSetEvent, KeRaiseIrql etc. which are excluded from localized-executions with skipping. However, analysis range 4 has negligible improvement since it does not involve any calls to standard code-streams. Figure 6.6d shows the performance of Cobra with block-coalescing and skipping on standard as well as already analyzed malware code-streams. As an example, the single-step handler of W32/Ratos always invokes a procedure that handles a variant of the TEA decryption algorithm, several times within its code-stream. This procedure never changes in terms of its semantics and can be skipped after analyzing it once. As seen from the graph of Figure 6.6d, analysis ranges 1–4 have a reduced latency from their counterparts in Figure 6.6c. Note

that analysis range 4, which showed a negligible change with skipping of standard code-streams, shows a noticable latency reduction with skipping applied to already analyzed malware specific code-streams.

Thus we can conclude that the performance of the framework is highly dependent on the nature of code being analyzed and the style of analysis employed by an individual (in terms of selecting analysis ranges, coalescing blocks, choosing code-streams to skip etc.). However, the measurements give a gross indication of the latency one might expect. As seen from Figure 6.6, even the worst case latency (without block-coalescing and skipping) of the framework is found to be within limits to suit interactive analysis.

6.6 Summary

In this chapter we have presented Cobra, the subsystem of WiLDCAT that is responsible for empowering the environment with fine-grained malicious code analysis capabilities. Cobra overcomes the shortcomings in current research involving fine-grained code analysis in the context of malware by providing stealth and efficient fine-grained analysis techniques. Cobra does not make any visible changes to the executing code and hence cannot be detected or countered. The framework can capture multithreaded, SM-SC code and obfuscated code in both user- and kernel-mode while incurring a performance latency that is suitable for interactive analysis. The framework supports selective isolation – a technique that enables fine-grained analysis of malware specific code-streams while co-existing with normal code-streams in real-time.

CHAPTER 7

COARSE-GRAINED MALWARE ANALYSIS

In this chapter we discuss the design, implementation and performance of SPiKE, the subsystem of WiLDCAT, that incorporates novel techniques empowering the environment to perform coarse-grained code-analysis at runtime on malware code-streams.

7.1 Introduction

The first and a very important step in the analysis process, involves monitoring malware behavior with respect to the system, to construct an execution model of the malware which aids in further finer investigations. As examples, the W32/MyDoom [44] and the W32/Ratos [74] propagate via e-mail and download and launch external programs using the network and registry. Such behavior, which includes the nature of information exchanged over the network, the registry keys used, the processes and files created etc., is inferred by employing coarse-grained analysis on malware specific code and related services of the host operating system (OS) pertaining to process, network, registry, file etc. Once such behavior is known, fine grained analysis tools such as debuggers are employed on the identified areas to reveal finer details such as the polymorphic layers of the trojan, its data encryption and decryption engine, its memory layout etc.

Monitoring malware behaviour with respect to the OS entails modification of the OS, supporting applications and/or the malware for possible semantic alteration — a technique that is called instrumentation.

There has been a plethora of research involving instrumentation, however those that are applicable to coarse-grained malware analysis rely on binary instrumentation (instrumentation without the need for sources).

Offline binary instrumentation (used by tools such as EEL [36] and Etch [57]) involves rewriting program binaries in order to obtain control at specified points during runtime. Probe-based binary instrumentation (used by tools such as Detours [30] and Sakthi [77]) involves runtime modification of binary code so as to enable a replacement code to be executed when execution is transferred to a specified point in the original code. Just-in-time compilation (JIT) is another type of binary instrumentation which employs a virtual machine approach to gain control at specific points. Examples include Pin [41] and and DynamoRIO [11]. Table-based binary instrumentation changes addresses of functions within the host OS system table in order to gain control when a specific OS function is invoked (used by tools such as STrace [70] and Regmon [62]). Filter-based binary instrumentation employs filter device drivers in order to gain control when a request is made to the target device (e.g filesystem). This technique is used by tools such as FileMon [61] and TDIMon [56]. Hardware-based binary instrumentation relies on processor supported breakpoint ability to gain control at specific points during execution of code. This technique is generally used by debuggers such as WinDBG [29] and GDB [42].

All the above methods have severe shortcomings in the context of malware analysis due to their common design principle: they are designed for normal program instrumentation.

Offline and Probe-based binary instrumentation are obtrusive as they involve modifying the target code (either before or during runtime). Most if not all malware are sensitive to code modification and employ techniques such as polymorphism and metamorphism to change their own code during runtime. Table and Filter-based binary

instrumentation are not comprehensive as they can only be used to instrument a subset of the host OS functions. Hardware and JIT based instrumentation are unobtrusive. However, current hardware only support a very few instrumentation points (as low as 2–4) to be active simultaneosly which is a serious restriction while analyzing complex functionalities contained in current generation malware. JIT-based binary instrumentation does not contain adequate support for multithreading and self-modifying and/or self-checking (SM-SC) code and are unable to analyze code executing in kernel-mode. Further, JIT-based binary instrumentation is more suited for fine-grained instrumentation and is an overkill in terms of latency for coarse-grained analysis. Also, recent trends in malware show them using instrumentation for their own functioning therby overriding existing instrumentation and increasing anti-analysis methods, rendering these binary instrumentation strategies severely limited in their use.

SPiKE overcomes the shortcomings of current coarse-graind code analysis techniques in the context of malware. The framework is based on a core design principle:

```
provide support for any form of malicious coding technique
  while ensuring complete control over instrumentation
```

SPiKE provides unobtrusive, persistent and efficient coarse-grained instrumentation, supporting multithreading and SM-SC code in both user- and kernel-mode. The instrumentation deployed by SPiKE is *unobtrusive* in the sense that it is completely invisible and cannot be detected or countered. It is persistent in the sense that the framework is always the first to get control even if an executing code-stream deploys its own instrumentation over SPiKE. The framework achieves *efficient* instrumentation using special techniques such as redirection and localized-executions.

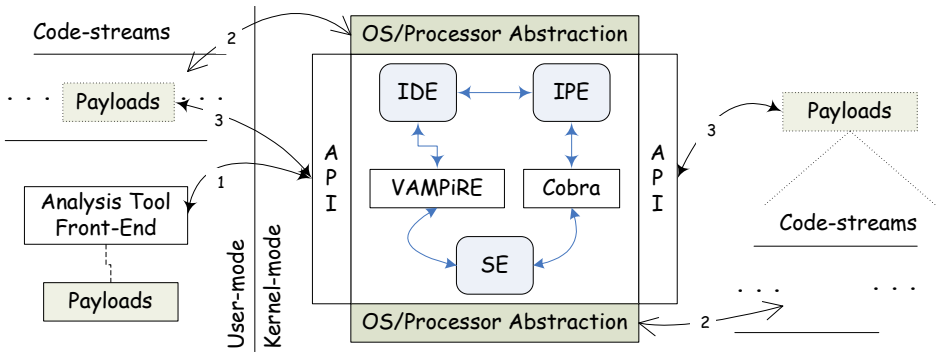


Figure 7.1. Architecture of Spike.

7.2 Overview of SPiKE

SPiKE supports instrumentation of both code constructs and/or data locations. Instrumentation under SPiKE is facilitated by a technique that we call Invisi-drift. The basic idea involves inserting what we call drifters, at memory addresses corresponding to code-constructs and/or data locations for instrumentation purposes. Drifters are invisible logical elements that trigger by responding to reads, writes and/or executes to their corresponding memory addresses. When a drifter triggers, SPiKE gets control and invokes the corresponding instrument. Instruments can then monitor and/or alter the executing code-stream as desired before returning control. The framework also allows an instrument to invoke the original construct at the drifter location. This allows an instrument to rely on the original code to perform the meat of the functionality while adding minor semantic changes.

SPiKE also employs a technique that we call redirection, to improve the performance in cases where instrumentation is deployed on functions housed in standard libraries. This is achieved by a technique we call scattering which involves restructuring the functions within the standard libraries in binary to minimize latency.

Figure 7.1 illustrates the current architecture of SPiKE. The framework core consists of: (a) a Instrumentation Deployment Engine (IDE), (b) a Instrumentation Per-

sistence Engine (IPE) and, (c) a Stealth Engine. The IDE and IPE employ VAMPiRE and Cobra respectively for a part their implementation. The framework core resides in kernel-mode allowing it to have control over both user- and kernel-mode code-streams.

As Figure 7.1 shows, there are typically three elements present during an analysis session: the target code-streams, the analysis tool front-end, and the tool payloads. The tool payloads contain instruments for constructs that are instrumented in the target code-streams. Both the tool front-end and tool payloads employ SPiKE's API to interact with the framework core. The tool front-end initialises the framework and registers the tool-payloads with the framework. The framework and the tool-payloads set up instrumentation on desired constructs on the target code-streams. The IDE is responsible for inserting drifters at desired locations and handles their triggering. It is also responsible for ensuring that a drifter instrument is loaded into the target code stream upon triggering. The IPE is employed to efficiently tackle issues involving reads, writes and/or executes to memory locations with drifters, thereby ensuring instrumentation persistence while co-existing with SM-SC code, code obfuscations and instrumentation strategies employed by a malware. The SE is responsible for shielding the framework against any form of detection or countering.

SPiKE is completely re-entrant, as it does not make use of any OS specific routines within its core components and uses shared memory with its own isolation primitives for interprocess communication. The framework also allows the tool payloads to communicate with the tool front-end in real time.

7.3 Stealth Instrumentation using Invisi-Drift

The framework inserts, what we term drifters, at memory addresses corresponding to code constructs and/or data locations, whose instrumentation is desired. A drifter is a logical element, that is a combination of a breakpoint and an instrument and can be

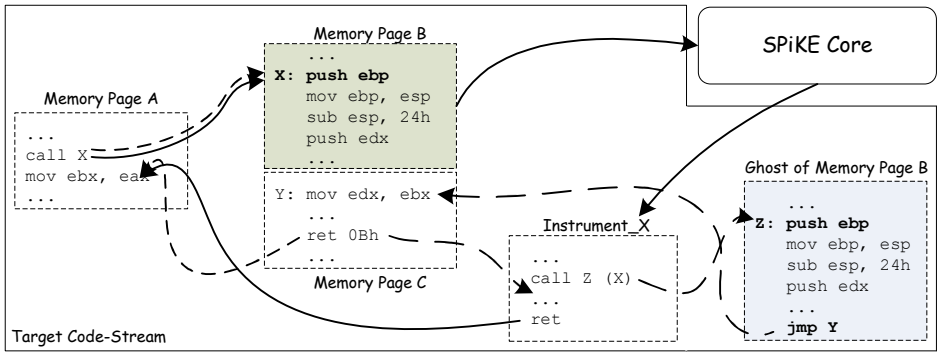


Figure 7.2. Drifters and Stealth Instrumentation.

active or inactive at any instant. The breakpoint provides the ability to stop execution of a code-stream, at runtime, at a desired memory address or on accesssing a given memory or I/O address. The instrument, is a user-defined function to which control is transferred when the breakpoint is reached. A drifter is classified into read, write and/or execute and is said to trigger when it responds by invoking the instrument on reads, writes and/or executes to the memory address where it is inserted. Execute drifters are usually used to instrument code constructs and are inserted at memory address corresponding to instructions, while read and write drifters are usually used to instrument data locations and can span a range of a desired number of bytes in memory. SPiKE makes use of VAMPiRE, to set read, write and/or execute breakpoints for the respective drifter types.

Let us consider Figure 7.2, which shows part of three memory pages in a target code-stream. Memory page A contains an invocation to a function X which is in memory page B. Further, let us assume that function X continues on memory page C which immediately follows memory page B. Also, let us assume that X is instrumented and the instrument is Instrument_X.

When an active drifter (in this case an execute drifter) is deployed on a memory address (in our example the start of function X), SPiKE first makes a copy of the entire memory page corresponding to the memory address. This copy is what we call a drifter

ghost. The drifter ghost is used for two purposes: (a) to allow instruments to invoke the original construct (more details follow) and (b) for instrumentation persistence (see Section 7.4).

After creating the drifter ghost, the framework sets a stealth breakpoint on the memory address whose type corresponds to the drifter type (in our example an execute breakpoint). When the drifter triggers, the instrument corresponding to the drifter is invoked (in our example Instrument_X), which can then monitor and/or alter program behavior by overwriting registers and/or memory (including the stack), before returning to the caller.

As seen from the solid and small dashed lines in Figure 7.2, the execution flow and memory contents from the perspective of the code-stream invoking an instrumented code construct remains unchanged with or without instrumentation.

Instruments can also invoke the original code at the drifter location within themselves. This feature could be used to chain to the original construct to do the meat of the processing while employing changes to the result. In our example, Instrument_X can invoke X with a different set of parameters than the original in order to mark memory regions so that execution of code within them can be monitored. SPiKE makes use of the drifter ghost for the purpose of original construct invocation. The drifter ghost for standard code-streams is constructed by relocating the code in the original memory page. This ensures that relative branche addresses remain the same when code is executed within the drifter ghost. In our example, when Instrument_X invokes the original construct via SPiKE, it will in reality execute code within the drifter ghost and not the original memory page corresponding to function X. Thus, the original memory page attributes always remains not-present, ensuring future drifter triggerings. This is especially true with multithreaded code where an instrumented code can be invoked while the instrument is still processing.

The drifter ghost is constructed to chain to the code within the following memory page of the original memory page. This ensures the normal semantics of the original construct when invoked from an instrument in case the original construct spans multiple memory pages. In our example, the drifter ghost of memory page B chains to Memory page C which is the remainder of function X. In cases where instrumentation is deployed on non-standard code-streams or when multiple drifters are set on the same memory pages, the framework employs the concept of stealth localized-executions to achieve original construct invocation (see section 7.4 and section 7.5).

The instrument can return control to the target code-stream either by simply returning via SPiKE or by chaining to the original code stream. In both cases, the framework removes any traces of itself (instruments, ghosts and framework specific code/data) from the target code-stream before transferring control. This is required for the framework stealthness (see Chapter 9).

The instrumentation model of SPiKE is much more flexible than the ones found in current dynamic binary instrumentation techniques in that: (a) it decouples the instrument from the construct being instrumented, which enables the instrument to invoke the original construct as many times as required with different sets of parameters on the call stack and (b) it is a generic method that can capture invocations to constructs that are not pure functions and (c) it can instrument even data locations. This feature can be used to monitor access to critical structures such as process and/or, thread environment blocks, exception chains etc.

Instruments always run in the mode that caused the drifter to trigger (user- or kernel-mode) with interrupts enabled. Reads and/or writes to data on the target code-stream are always synchronized. Thus instruments have no restrictions in invoking any API from within themselves and are multithread safe. Instruments also have the ability to insert and/or remove drifters at any memory location dynamically.

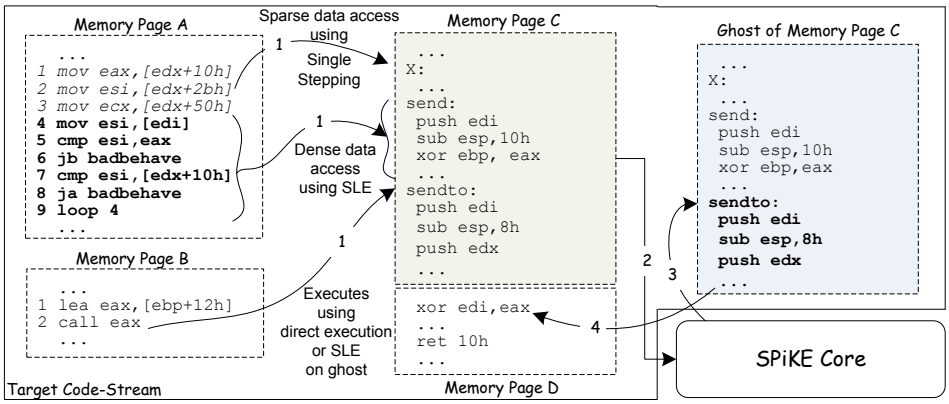


Figure 7.3. Instrumentation Persistence.

7.4 Instrumentation Persistence

When a drifter is inserted at a desired memory location, the entire memory page corresponding to the location, has its attribute set to *not-present* as described in the previous section. Further, for instrumentation persistence, such memory pages always need their attribute set to *not-present* for the lifetime of a drifter. These facts give rise to a couple of issues that need to be dealt with. The first is when the target code reads and/or writes to drifter locations or to locations within the memory page containing drifters. The second situation is when a target code-stream executes code on the memory page with an active drifter, but not exactly at the drifter location(s). Both cases result in PFEs due to the destination memory page attribute being *not-present*. SPiKE employs Cobra, to eliminate such PFEs and while ensuring instrumentation persistence.

Let us consider Figure 7.3, which shows code fragments from our analysis sessions involving the W32/MyDoom. The code fragments shown in memory pages A and B represent part of the virus which is responsible for a trojan download while memory pages C and D show portions of the virus localized DLL for socket API functions (see Chapter 10 for more details on our experience with W32/MyDoom). We set an execute drifter at the send function of the localized DLL in order to monitor the download.

In situations involving sparse read and/or writes to memory pages containing active drifters, SPiKE makes use of VAMPiRE's breakpoint persistence feature to ensure instrumentation persistence. Let us consider line 2 of the code fragment of memory page A in Figure 7.3. It performs a read from memory page C which result in PFEs due to memory page C being not-present due to the execute drifter at connect. VAMPiRE employs a single-step handler for breakpoint persistence in such cases. The single-step handler sets the memory page attribute back to its original, executes the instruction causing the PFE and sets the memory page attribute to not-present to trigger future breakpoints.

Let us consider lines 4–9 of the code fragment in memory page A in Figure 7.3. This code fragment is responsible for decryption of the send function within the localized DLL before it is executed. It writes to the memory locations following the send function in memory page C using a loop and hence results in multiple PFEs.

When there is dense data access on a memory page containing active drifters (in this case, lines 4–9 of the decryption code in memory page A accesses portions of memory page C with an execute drifter), SPiKE uses the resulting PFE to begin SLE at the instruction performing the read/write. Once the blocks for the target code-stream are created, the framework executes them locally while setting the attribute of the memory page containing the drifters to its original. This prevents multiple PFEs that would have otherwise occured if using VAMPiREs feature of breakpoint persistence.

With reads and writes to memory pages containing active drifters, SPiKE does not commence SLE on the intial PFE. Instead the framework uses heuristics to determine the efficacy of SLE. As an example, the loop in lines 4–9 of the code fragment in memory page A of Figure 7.3 is found to iterate on an average of 100–150 times from our analysis sessions. The framework refrains from starting localized-executions until a few PFEs that occur during the initial iterations of the loop. Once SPiKE determines that the PFEs

are occuring due to a localized region of code as in the example, the framework starts SLE to minimize PFEs in the future iterations. This scheme ensures that SLE are not undertaken for sparse data access since that would amount to an overhead more than the latency of the PFE and single-step exceptions themselves.

SPiKE sets the memory page attribute containing active drifters to not-present and stops localized executions upon termination of the identified range of code performing the data access. When a future PFE occurs via a target code-stream for which SPiKE has already constructed blocks, the framework automatically executes the coalasced version of the blocks corresponding to the target code-stream directly from the block-repository, thereby eliminating the need to create blocks once again.

SPiKE also employs SLE for situations where there are executes to memory page containing active drifters. The only difference is that SLE operates on the drifter ghost instead of the original memory page. Let us consider memory page C of Figure 7.3 which also contains part of the sendto function of the localized DLL. Thus, when a target code stream executes the sendto function (lines 1–2 in memory page B , Figure 7.3), it would result in a PFE due to the page being not-present. SPiKE then starts SLE on the drifter ghost and stops SLE when execution branches out of the drifter ghost, or if execution reaches a point in the drifter ghost which corresponds to another active drifter in the original memory page. This allows SPiKE to tackle multiple drifter triggering within the same memory page. Thus, in our example, SLE of sendto would fall through to send (since sento uses send) which can trigger the drifter at send (also see section 7.5).

7.5 Selective Indirection

With original construct invocation, an instrument of a target construct might be invoked by the framework even though the target construct was not invoked directly by the executing code-stream.

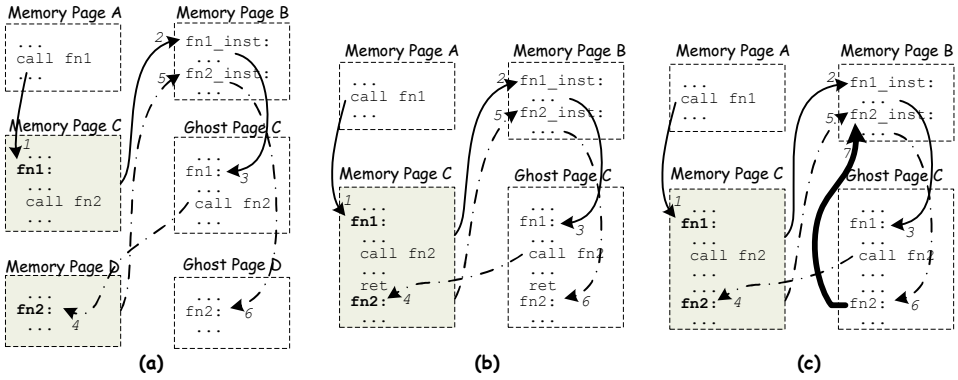


Figure 7.4. Selective Indirection: (a) Dependent function invocation on different memory pages, (b) and (c) Dependent function invocation on the same memory page.

Let us consider Figure 7.4a which shows code fragments in various memory pages alongwith some instruments and drifter ghosts. Function fn1 is invoked by function f0 at some point. Further, function fn1 at some point invokes function fn2 to accomplish a part of its functionality. Let us assume that functions fn1 and fn2 are instrumented with fn1_inst and fn2_inst being their instruments respectively. Also, let us assume that fn1_inst and fn2_inst invoke the original functions fn1 and fn2 from within themselves.

Let us consider the flow of execution shown in steps 1–3 using solid arrow lines in Figure 7.4a. When function fn1 is invoked, the drifter associated with it triggers and invokes fn1_inst. Then fn1_inst invokes the original function fn1 from within itself at some point. Let us now consider the flow of execution shown in steps 4–6 using dashed-dotted arrow lines in Figure 7.4a. When fn1_inst invokes fn1, it results in a call to function fn2 which results in fn2_inst being invoked.

This semantic might be undesirable in most cases as one only wishes to instrument APIs in their pure calls. As an example, the socket API sendto invokes connect, send and close APIs at some point within its execution. When sendto is instrumented, the instrument will see all the parameters it needs to, since the invocations to connect, send and close within sendto use a subset of the input parameters to sendto. Thus

instrument invocations of connect, send and close when invoked from sendto is clearly not consequential.

SPiKE can be configured to recognize such dependencies and in such cases will suppress such dependent instrument invocations during original construct invocation. When a drifter triggers, SPiKE obtains the value of the memory address of the instruction that causes drifter triggering. If this memory address belongs to an instrument code, the framework knows that the drifter triggering is due to a dependency and not for a direct invocation. This is because, with the instrumentation mechanism of SPiKE, the original construct at a drifter location is always invoked only by the instrument (either for chaining or for processing). Once, the framework determines that a drifter triggering is due to a dependent invocation, it can (depending upon the configuration) directly invoke the drifter original construct from the drifter ghost instead of its instrument thereby suppressing triggering of dependent functions that are instrumented. We call this mechanism selective indirection.

Let us consider Figure 7.4b where the functions fn1 and fn2 share the same memory page. When function fn1 is invoked, the drifter associated with it triggers and invokes fn1_inst. Then fn1_inst invokes the original function fn1 from within itself at some point. This is shown in steps 1–3 using solid arrow lines in Figure 7.4b. If SPiKE is setup for selective indirection (which is the default), the framework will directly execute code within the drifter ghost. This automatically supresses dependent instrument invocations. In this case the invocation to fn2 from within the drifter ghost of memory page C will result in fn2 within the drifter ghost getting control (since CTIs within drifter ghost is relocated) and hence will not result in a drifter triggering. However, if SPiKE is not setup for selective indirection, the framework will employ SLE on the drifter ghost rather than execute the code directly. This allows the framework to trigger the drifter at function

fn2 during SLE using events. This is shown in steps 4–6 using dashed-dotted arrow lines in Figure 7.4b.

Figure 7.4c shows a special case of Figure 7.4b. The only difference being that fn2 is actually a part of fn1. With selective indirection, the flow of execution is as described in the previous paragraph for Figure 7.4b. Without selective indirection, the execution flow is the same until step-6 of Figure 7.4b. However, at step-6 when fn2_inst invokes the original function fn2, it will result in fn2_inst being invoked again due to SLE triggering an event on reaching fn2 (shown as a thick solid arrow line in Figure 7.4c). Thus, in this case fn2_inst will be invoked in a cyclic fashion. Solutions to cases like this varies and is based on the dependency between the functions. In this case, a solution is to execute the function fn2 directly without employing SLE.

7.6 Redirection

Drifters incur latency in certain situations where code within the same memory page as the drifter needs to be executed. This is due to SLE on the drifter ghost as discussed in the previous section. This latency can be nullified for execute drifters that are inserted on OS and supporting functions housed within dynamic link libraries (DLL) or shared libraries. The reduction in the runtime latency can be substantial if one considers the fact that most of the services offered by the OS or an application in contemporary OSs, are in the form of dynamic link libraries (DLL) or shared libraries. While there are programs that make use of static libraries, their numbers are very limited (about 1% of all applications in our experiment). Also on some OSs such as Windows it is simply not possible to statically link against the system libraries without severely compromising compatibility.

Every DLL or a shared library exports the functions contained within it using an *export-table*. Among other fields, the export-table contains an array of pointers to func-

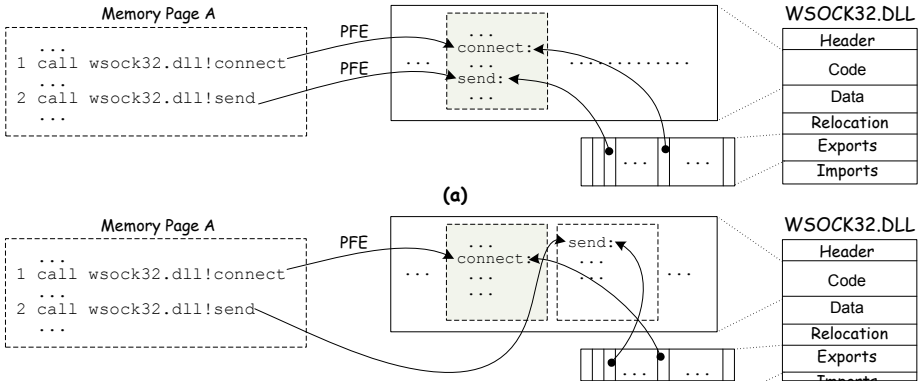


Figure 7.5. Redirection.

tions within the library known as the *Export Address Table* (EAT). When an application uses a DLL or a shared library, it *imports* a set of functions from it. This is in the form of an *Import Address table* (IAT) or a *Procedure Linkage Table* (PLT) within the application. Further details regarding the executable formats of DLLs and shared libraries can be found in the Portable Executable Specification [49] and the Executable and Link Format Specification [75]. The OS loader, obtains a list of the functions that are being imported, loads the appropriate libraries, fetches the addresses of the functions using the EAT and updates the IAT or the PLT within the application. Figure 7.5a shows a sample function invocation from an application using the DLL, `WSOCK32.DLL`.

SPiKE re-writes the DLL or the shared library that contain the function(s) to be instrumented, such that every function starts on a unique memory page (this might not always be possible, see section 7.6.1). Further, the re-writing is carried out in such a fashion that that memory page contains portions of none of the other functions. The rationale behind this approach is that drifters on standard functions are usually set on the start of the function. Thus, with our approach, there will be no executes to memory page containing active drifters apart from the function invocation itself.

Figure 7.5a and Figure 7.5b shows a portion of the WSOCK32.DLL containing the connect and send functions before and after the re-writing process respectively. The two functions share the same memory page in the original DLL. Thus, if the connect function is instrumented in the original DLL, invocations to send will incur latency due to drifter triggering. However, with the re-writing, as seen in Figure 7.5b the connect and the send function are rewritten such that they fall on different memory pages within the DLL, ensuring that instrumentation of connect does not impact the invocation of the send. We call this mechanism as redirection (as opposed to directly inserting a drifter at a memory location, which we call direct instrumentation).

The re-writing is carried out in binary (usually a one time process before analysis) and does not require the sources and/or any debug symbols for the library. The export-table entries of the module are then updated to point to the new addresses of the functions.

The re-writing process can result in some extra instructions being embedded in addition to the normal set of instructions within a function. The purpose of these extra instructions – called bridge instructions – is to ensure the original semantics of a function in situations where a function needs to be split up the re-writing process. The re-writing process also inserts extra memory-pages within the target library in order to ensure that most of the exported functions fall on different memory pages. These extra memory-pages are called fillers.

7.6.1 Scattering

With redirection, a straightforward scheme is to allocate one function per memory page. However, this might result in a lot of fillers being employed which results in an increase in size of the library which can make the framework prone to detection. As an example, if one considers KERNEL32.DLL, a dynamic library under the Windows OS, it

exports about 1024 functions. The scheme of having every function on a different memory page results in an increase of 400kb to the original library which is an increase that is not normal (even with dynamic updates and service packs). To reduce the memory footprint as a result of redirection, SPiKE allows tuning the number of functions per memory page.

When there are multiple functions per memory page, the runtime performance of redirection varies depending on the way in which the functions are coupled together in a single memory page. Let us consider KERNEL32.DLL a system DLL under the Windows OS. Redirection on KERNEL32.DLL results in multiple functions that need to be allocated the same memory page since adding too many fillers would work against the framework stealthness (see Chapter 9). Let us assume that file APIs open, write and close are put into the same memory page. Let us further assume that the open API is instrumented. This results in latency due to executes to write and close. The overhead can be substantial if the write API happens to be within a loop (this can be the case memory contents are committed to a file).

SPiKE can recognize such situations and will ensure that the redirection-pads for the APIs belonging to a group (in this example, open, read, write close are all file related APIs from kernel32.dll) occupy different memory pages when possible. We call this technique scattering. The rational idea behind this approach is that APIs belonging to a group are more likely to be executed in a local region than those with different groups. (example being opening a file, writing it and then closing it). For situations where redirection pads for APIs from a same group share a memory page, SPiKE assigns priorities to APIs such that more frequently used APIs are scattered in preference to others. The framework also allows the user to override existing scatter pattern to cater to desired needs.

Code Stream & Instrumentation Strategy	Strategy for Original Construct Invocation	Strategy for Executes to Memory Page with Active Drifters
K-mode + Non-standard + Direct + SI	SLE	SLE
K-mode + Non-standard + Direct – SI	SLE	SLE
U-mode + Non-standard + Direct + SI	SLE	SLE
U-mode + Non-standard + Direct – SI	SLE	SLE
K-mode + Standard + Direct + SI	Direct Execution	Direct Execution
K-mode + Standard + Direct – SI	SLE	SLE
U-mode + Standard + Direct + SI	Direct Execution	Direct Execution
U-mode + Standard + Direct – SI	SLE	SLE
K-mode + Standard + Redirection + SI	Direct Execution	Direct Execution
K-mode + Standard + Redirection – SI	SLE/Direct Execution	SLE/Direct Execution
U-mode + Standard + Redirection + SI	Direct Execution	Direct Execution
U-mode + Standard + Redirection - SI	SLE/Direct Execution	SLE/Direct Execution

K-mode = Kernel-mode, U-mode = User-mode, SI = Selective Indirection, SLE = Stealth Localized-executions

Figure 7.6. Execution Strategies on a Drifter Ghost.

7.7 Ghost Executions

As discussed in the preceeding sections, there are two situations which can result in the execution of code within a drifter ghost: (a) when an instrument wishes to invoke the original construct at the drifter location, and (b) when a target code-stream executes code on the memory page with an active drifter, but not exactly at the drifter location(s). The framework employs a combination of direct execution and SLE on the drifter ghosts depending upon the nature and the executing privilege level (user- or kernel-mode) of the code-stream, the instrumentation strategy used and selective indirection. Figure 7.6 shows the default strategies employed by SPiKE for executions on a drifter ghost under all possible situations.

A point to note from Figure 7.6 is that executions on a drifter ghost of a non-standard code-stream is always performed using SLE. This is needed to retain control in kernel-mode as well as for supporting SM-SC code or any form of code obfuscations. Also note that with redirection, in certain cases, the framework can employ direct execution of the drifter ghost without selective indirection while still being able to trigger dependent invocations. This can be done in cases where a memory page contains only a single active drifter or multiple drifters whose functions have no dependency. As an example, let us consider the sendto and send socket API functions within WSOCK32.DLL. Let us

assume that both of them are instrumented. Direct instrumentation will need to use SLE without selective indirection in order to ensure that an invocation to sendto also triggers the invocation to send (since sendto uses send). However, in redirection, with scattering, send and sendto fall on different memory pages. Hence, redirection can employ direct execution on the drifter ghost of sendto and still be able to trigger the drifter upon invocation to send (see section 7.5).

7.8 Performance

In this section, we present the performance of SPiKE with and without instrumentation. We also compare SPiKE with current dynamic binary instrumentation frameworks and show that the framework provides features that are highly conducive to malware analysis while incurring latencies which are well within limits to suit interactive analysis.

Before we proceed to discuss the performance of the framework, a few words regarding the test-bench are in order. Our experimental setup consists of an Intel Xeon 1.7 Ghz processor with 512MB of RAM running Windows and Linux. We use processor clock cycles as the performance metric for our measurements. This metric is chosen, as it does not vary across processor speeds and also since it is a standard in literature related to micro-benchmarks. The RDTSC instruction is used to measure the clock cycles. We use WatchDog, our analysis tool (see Chapter 4) for performance measurements.

Coming up with representative analysis session for purposes of performance evaluation, is not an easy task since it depends on a lot of factors, chief among them being the method of analysis adopted by an individual, the structure of the executing code and dependency of one construct over the other, which are not easily characterized. Thus, we will concentrate on presenting the performance of the framework related to specific analysis sessions with the W32/MyDoom and W32/Ratos. The performance of the framework for other situations can be estimated in a similar fashion.

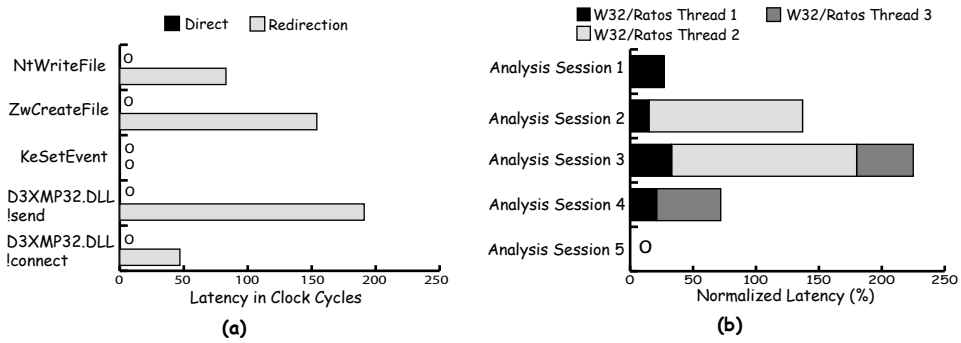


Figure 7.7. Performance Without Instrumentation: (a) Per-construct and (b) Per-System.

Unless otherwise specified, for the graphs encountered in the following sub-sections, the x-axis is the amount of extra clock cycles that are incurred as opposed to the native run-time of the function. Also, parts of some graphs are magnified (indicated by dotted lines) to provide a clear representation of categories with low values on the x-axis.

7.8.1 Latency Without Instrumentation

Figure 7.7 shows the framework latency without instrumentation. SPiKE incurs no latency without instrumentation for direct instrumentation. However, with redirection based instrumentation, there is a very small runtime overhead without instrumentation because of the execution of bridge instructions within the redirected libraries. This is shown in Figure 7.7a.

When there are non-standard kernel-mode code streams, SPiKE incurs a runtime overhead without instrumentation for both direct and redirection based instrumentation. This is because the framework SE employs SLE on non-standard code streams in kernel-mode for the framework stealthness. The actual latency depends on the nature of the non-standard code streams. Figure 7.7b shows the normalized latency of various malware specific kernel-mode threads from our analysis sessions (AS) involving the W32/Ratos. As seen AS-2 incurs a higher latency than AS-1. This is due to the presence of heavy self-

modification techniques within the code-streams of W32/Ratos Thread 2 when compared to Thread 1. Further, AS-3 incurs the highest latency due to the execution of all three threads of the W32/Ratos. In general, the higher the number of non-standard kernel-mode code-streams, the greater the latency without instrumentation. Note that AS-5 which has no non-standard kernel-mode threads, incurs no latency.

7.8.2 Latency With Instrumentation

SPiKE's latency due to active direct and redirection based instrumentation can be divided into: (a) latency due to instrument invocation, and (b) latency due to reads, writes and/or executes to a memory page containing active drifters.

7.8.2.1 Instrument Invocation Latency

The instrument invocation latency can be further split up into latency due to: (a) instrument control gain - the time that has elapsed after the transfer of control to a code construct, and before control is handed over to its corresponding instrument, (b) original construct invocation - the extra time that has elapsed in executing the original construct from within the instrument, than what would have been normally without instrumentation, and (c) instrument control release - the time that has elapsed after the instrument returns and before control is transferred to the target code-stream.

Figure 7.8 shows the latency involved in instrument invocations for both direct and redirection-based instrumentation of SPiKE, for various standard and non-standard functions from our analysis sessions involving the W32/MyDoom. Note that in the graphs, the measurements for redirection based instrumentation in the context of non-standard code streams is indicated by a NA (Not Applicable) since redirection cannot be applied to non-standard DLLs without compromising on the framework stealthness.

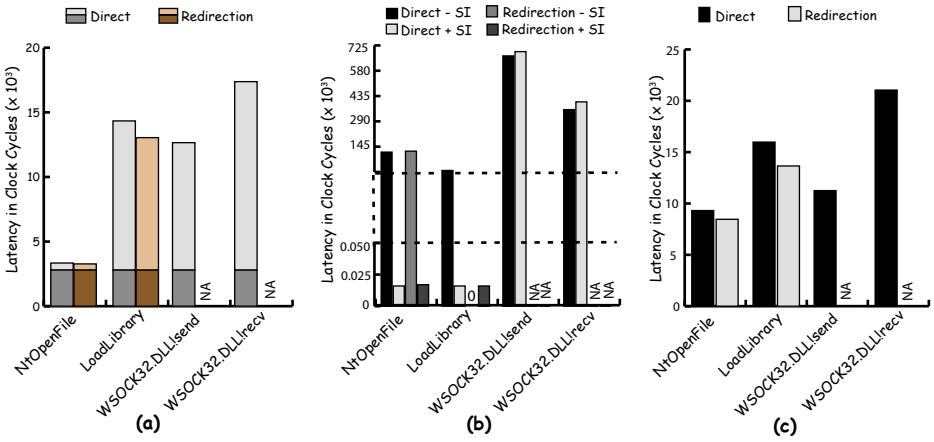


Figure 7.8. Instrument Invocation: Latency due to (a) Instrumentation Control Gain, (b) Original Construct Invocation, and (c) Instrumentation Release.

Both direct and redirection based instrumentation incur similar instrumentation control gain latency in both user- as well as kernel-mode (shown in Figure 7.8a). This is due to drifter triggering (which incurs a constant processing overhead due to the PFEs) and payload deployment which depends on the number of active instruments within a payloads and their size.

Figure 7.8b shows the latency due to original construct invocation. Direct instrumentation of non-standard code-streams in both user- and kernel-mode incurs high latency for original construct invocation as SPiKE employs SLE to support SM-SC code and to ensure control in kernel-mode (send and recv functions within localized WSOCK32.DLL). Direct and redirection-based instrumentation of standard code-streams with selective indirection in both user- and kernel-mode have a very low latency due to direct execution of code within the drifter ghost when there is original construct invocation (NtOpenFile and LoadLibrary). However, without selective indirection, the direct method incurs a higher latency due to SLE being employed for ghost executions. The latency of original construct invocation of redirection without selective indirection depends upon scattering and the number of drifters on a single memory page (see section 7.5).

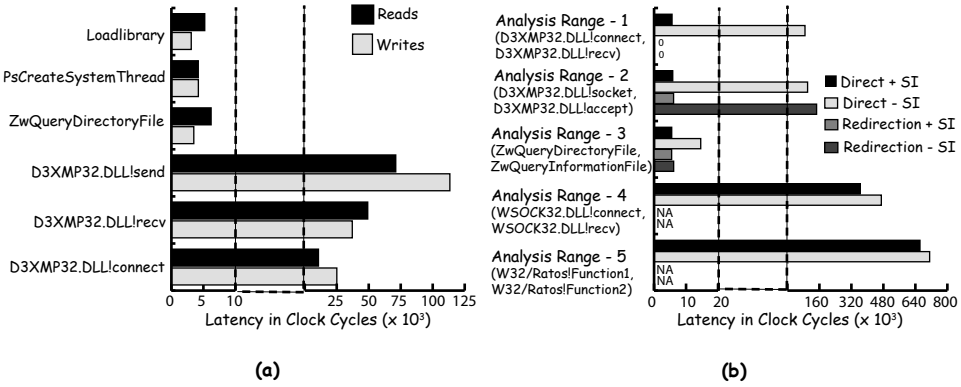


Figure 7.9. Latency due to: (a) reads and/or writes, and (b) executes to memory page containing active drifters.

Thus, for Loadlibrary the latency is zero (since for the analysis session, Loadlibrary shared its memory page with two other functions that had no dependencies between themselves) while for NtOpenFile the latency for original construct invocation is much higher as it shared its memory page with NtCreateFile, which it invoked from within itself. Thus the framework employs SLE on the drifter ghost in order to trigger the drifter due to the invocation to NtCreateFile, which explains the overhead.

Instrumentation control release latency primarily depends on whether the release was due to a return from the instrument or due to a chaining to the original construct. In case of release due to return, the latency is minimal and is due to the framework removing traces of itself (shown in Figure 7.8c). In case of release by invoking original construct the latency incurred in addition to that shown in Figure 7.8c is similar to the latency incurred for original construct invocation as described earlier.

7.8.2.2 Other latency

The framework also incurs other forms of latency in the form of reads, writes and/or executes to a memory page containing active drifters.

Figure 7.9a shows the latency incurred by the framework due to reads and writes on certain system functions checked and instrumented by the W32/MyDoom and W32/Ratos and to functions housed in localized DLLs employed by the virus (in this case WSOCK-32.DLL), using both direct and redirection based instrumentation. Both methods incur latency due to the IPE using single-stepping and SLE for instrumentation persistence. One can observe from the graph that for functions housed in system and/or standard libraries, the number of reads and/or writes is very minimal (mostly the first few instruction of the function), whereas for a malware specific DLL that overwrites its own code, the latency is higher on account of multiple reads and/or writes due to its SM-SC nature.

Figure 7.9b shows the latency due to executes to memory pages containing active drifters. For each of the analysis ranges (AR), the first function has an active drifter while the execute is performed to the second function.

Direct instrumentation of non-standard code-streams in both user- and kernel-mode incurs high latency for executes to the memory page containing the drifters due to SLE employed by SPiKE to support SM-SC code and to ensure control in kernel-mode (AR 4 and 5). Direct instrumentation of standard code-streams with selective indirection in both user- and kernel-mode have a low latency due to the initial PFE and then direct execution of code within the drifter ghost(AR 1,2 and 3). However, without selective indirection, it incurs a high latency due to the employing SLE to ensure multiple drifter triggering on the same memory-page.

The latency of redirection-based instrumentation depends upon scattering. When there is a single function on a memory page, the latency is zero both with and without selective indirection. This is because the framework employs direct execution on the drifter ghost in either cases (AR 1, Figure Figure 7.9b where the connect function occupies a memory page by itself). When there are multiple functions on a memory page, the latency of redirection-based instrumentation depends on the dependency between the

functions. When the functions are not dependent (AR 3, Figure 7.9b), the latency incurred is very minimal both with and without selective indirection and is due to the PFE after which the framework employs direct execution on the drifter ghost. When the functions are dependent (AR 2, Figure 7.9b, the accept function uses the socket function within itself), the latency of redirection without selective indirection is much higher than with due to the framework employing SLE on the drifter ghost.

7.8.3 Memory Consumption

The direct and redirection based instrumentation strategies of SPiKE incur a memory overhead due to their design elements. The direct method has zero memory overhead without instrumentation. However, the memory overhead of redirection without instrumentation is due to scattering within a DLL/shared library which can result in filler pages. The exact memory overhead depends on the DLLs and the nature of code-streams within them. Figure 7.10a shows the memory overhead of redirection without instrumentation for several DLLs.

For active instrumentation using the direct or the redirection method, every memory page containing a drifter incurs a memory overhead of one extra memory page which is for the drifter ghost. Further, every instrument for an active drifter takes up additional memory space. The exact memory overhead in this case once again depends on the DLL/shared libraries that are scattered and on the number of drifters and the nature of their corresponding instruments. Figure 7.10b shows the memory overhead of an analysis session where all the functions of all the DLLs were instrumented and the all the instruments simply chained to the original function without any processing.

As seen, the total memory consumption is around 33MB which is not very demanding.

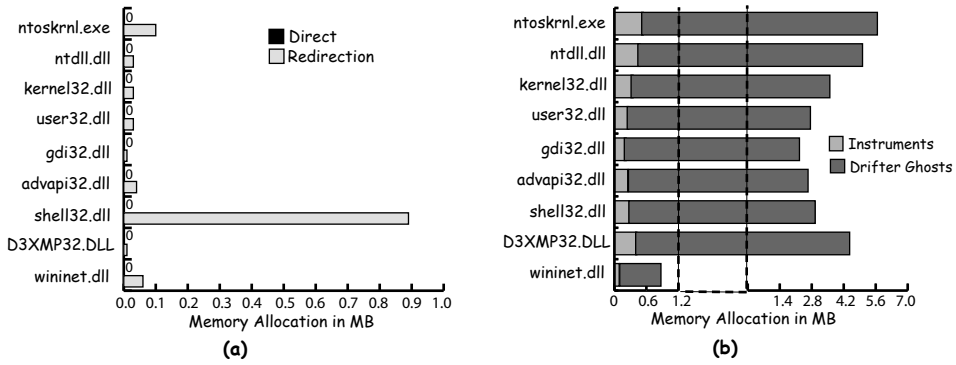


Figure 7.10. Memory Overhead: (a) Without Instrumentation, and (b) With Instrumentation.

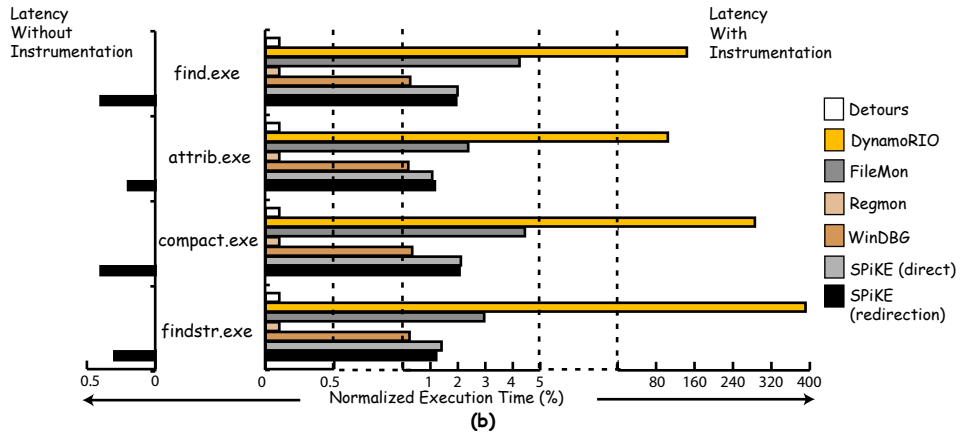
7.8.4 Framework Comparison

In this section we compare SPiKE with some popular dynamic binary instrumentation frameworks such as Detours, DynamoRIO, FileMon, RegMon and WinDBG. We chose these frameworks as our candidates for comparison due to their popularity and the fact that they allowed us to use the Windows OS as a common platform for comparision while providing a blend of currently available dynamic binary instrumentation strategies. We used the following release of each framework for comparison purposes: DynamoRIO 0.9.4, Detours 2.0, Filemon 4.34, Regmon 4.35 and WinDBG 6.6.7.5. We start by presenting a qualitative comparision of the frameworks followed by a quantitative comparison which presents the overhead involved in applying SPiKE's instrumentation in comparison to the other frameworks.

Figure 7.11a shows various features supported by SPiKE and how it compares to other existing dynamic binary insrumentation frameworks. SPiKE clearly stands out in terms of features supported in the context of malware such as transparency, stealthness, support for SM-SC and obfuscated code. However it also offers general features such as multithreading, original construct invocation, success rate, support for arbitrary constructs, that match and in some cases even better that of existing frameworks. Fur-

	Transparent	Original Construct Invocation	Selective Instru-mentation	Success Rate	Kernel Mode Support	Multi-threading	Arbitrary Code	Stealth & Persistent	SM-SC & Obfuscated Code	Number
WinDBG	✓	✗	✗	100%	✓	✓	✓	✗	✓	2 to 4
Detours	✗	✓	✗	< 100%	✓	✓	✗	✗	✗	Unlimited
Filemon	✓	✗	✗	100%	✓	✓	✗	✗	✗	Limited
Regmon	✗	✓	✗	100%	✓	✓	✗	✗	✗	Limited
DynamoRIO	✓	✗	✗	100%	✗	✗	✓	✗	✗	Unlimited
SPiKE	✓	✓	✓	100%	✓	✓	✓	✓	✓	Unlimited

(a)



(b)

Figure 7.11. Performance Comparison: (a) Qualitative, and (b) Quantitative.

ther, SPiKE also supports selective indirection, a feature that is not available in other frameworks.

For a quantitative comparision, coming up with a representative performance evaluation criteria was difficult since not all the features offered by SPiKE are available on current frameworks based on dynamic binary instrumentation. Further, the performance of dynamic binary instrumentation such as JIT-based depends on the nature of the executing code streams.

Thus, for a quantitative comparision we chose to instrument the file related APIs FindFirstFile and FindNextFile and measured the performance of these frameworks in the context of various real world applications using those APIs. The performance latency of the frameworks were computed as follows. First we measured the execution time of

the applications under all the frameworks without instrumentation. Then, we set instrumentation on the two file API's and reran the applications under all the frameworks and measured the execution time. To obtain the latency with instrumentation, we computed the normalized latency based on the difference in the execution times of two runs. Considering that SPiKE scores over all the frameworks in terms of the features provided and the fact that our main aim is to show that the SPiKE's instrumentation is suitable for interactive analysis, the performance criteria we have chosen for the quantitative comparison is acceptable.

Figure 7.11b shows the performance of SPiKE when compared to other dynamic binary instrumentation frameworks without any instrumentation. As seen only the redirection based instrumentation strategy of SPiKE incurs a minimal runtime overhead due to the bridge instructions.

With instrumentation, JIT-based frameworks such as DynamoRIO incur the highest runtime overhead. This is because they use a VM based approach to instrumentation. Filter-based frameworks such as FileMon incur the second highest runtime overhead. This is due to the fact that the host OS needs to propagate the filesystem device I/O request parameters through the device stack due to the filter. Probe-based and Table-based frameworks such as Detours and Regmon incur the least overhead as they mostly use a simple branch instructions for instrumentation purposes. Hardware-based frameworks such as WinDBG and SPiKE incur similar overheads. This is due to the triggering of the breakpoint and the page-fault exceptions respectively.

As seen from Figure 7.11c, SPiKE's performance is comparable to other dynamic binary instrumentation frameworks, but is not the most efficient. Given that the other dynamic binary instrumentation frameworks do not compare in capacity when it comes to malware analysis, the fact that SPiKE can achieve a latency close to these frameworks is acceptable.

7.9 Summary

In this chapter we have presented SPiKE, the subsystem of WiLDCAT that is responsible for empowering the environment with coarse-grained malicious code analysis capabilities. SPiKE overcomes the shortcomings in current research involving coarse-grained code analysis in the context of malware by providing stealth, persistent and efficient coarse-grained analysis techniques. The instrumentation deployed by the framework is completely invisible to the target code and cannot be detected or countered. Further, the framework allows a code-stream to deploy its own instrumentation over itself, while still ensuring that it is the first to get control. SPiKE supports multithreaded, SM-SC code and any form of code obfuscations in both user- and kernel-mode while incurring a performance latency well within limits for interactive analysis.

CHAPTER 8

DYNAMIC EXTENSIONS AND PAYLOADING

In this chapter we present techniques to dynamically monitor and/or enhance parts of the host OS without access to sources as well as load portions of the environment in the context of a specific process/thread under the host OS. Extenders — code constructs to enable extension of desired portions of the OS or supporting libraries in binary — is discussed. Payloading — a unified code injection mechanism in order to deploy the extenders and/or environment specific code in the context of a specific process or thread — is also discussed.

8.1 Introduction

Deploying WiLDCAT on a host OS calls for extending the behavior of several aspects of the host OS. For example, thread and process creation primitives on the host OS are employed by the environment in order to monitor and selectively deploy analysis mechanisms. Further, the strategies employed by the environment for malware analysis requires certain portions of the environment to be visible in the memory address space of a target process/thread.

While extending OS that are open source is trivial, the task becomes complicated with deployments that are commercial and binary-only.

Instrumentation is the ability to control constructs pertaining to any code. Constructs can either be pure (functions following a standard calling convention) or arbitrary (code blocks composed of instructions not adhering to a standard calling convention). By control we mean access to a construct for purposes of possible semantic alteration.

Consequently instrumentation comes in two flavors - Active and Passive. Active instrumentation involves altering the semantics of the construct whereas passive instrumentation retains the default semantics while performing other functions specific to the task in hand. Instrumentation can also be categorized into global instrumentation (where all processes see the instrumentation) or local instrumentation (specific to a process or a thread within the process). For a trivial example, consider instrumenting an operating system call such as the socket call to track open socket handles. This would amount to a Passive instrumentation, since we are not altering the default behavior of the socket call (which is allocate a socket handle), but merely tracking its allocation.

Instrumentation is not a new concept. Techniques of code patching have existed since the dawn of digital computing. There have been various improvements suggested by researchers such as executable image editing, executable and dynamic library import redirection, target function rewriting, dynamic compilation, object wrappers and so on. The next section briefly discusses a number of these works.

Most of these works are tailored towards a specific operating system and machine architecture, employing ad hoc techniques to deploy the instrumentation. Our framework code-named Sakthi, enables binary instrumentation across multiple operating systems and machine architectures. It is based on the concept of dynamic extensions and provides a code injection mechanism, which lends itself to implementation on a wide range of commercial and free operating systems. Additionally our framework is capable of providing options for global and local extensions, is able to extend pure or arbitrary constructs and deploy code in a new or an already executing process with minimal downtime.

8.2 Background

In a broad sense, research involving extending the OS are divided into how they achieve the addition or enhancements and how they deploy these additions or enhance-

ment at runtime. The following sections discuss the current techniques in the above two areas in detail.

8.2.1 Achieving Additions and/or Enhancements

8.2.1.1 Call Replacement in Source Code

One of the easiest and earliest methods of extension is replacing a call to a target construct in the program source, by a call to a replacement construct. The replacement construct then performs the intended task and then if need be calls the original target construct to accomplish the default functionality of the target construct.

This scheme however entails a program source rebuild as and when new constructs are extended. Also, this method relies on the availability of the sources to the program being extended.

8.2.1.2 Call Replacement in Object Code

This scheme is basically a variation of the previous method, the only difference being that the object code of the module or program is disassembled and the extended construct is put in place using a raw instruction assembler.

This method has one advantage from the previous in that, it does not rely on the sources to the program being instrumented. However, this method is static in nature and requires knowledge of the structure of the executable object code being instrumented.

8.2.1.3 Breakpoint Trapping

This scheme is very similar to the way in which debuggers work. A breakpoint instruction is inserted at the location of the target construct being instrumented. Thus, when the target construct is executed, a breakpoint exception is generated which is

subsequently handled to perform the required functionality pertaining to the construct. The breakpoint can be optionally re-inserted to enable instrumentation of future calls to the target.

This method works at the binary level and is dynamic. The only drawback to this method is its performance since every call to the construct that is extended, results in a processor breakpoint interrupt being generated.

8.2.1.4 Import Redirection

This method makes use of the import table (linkage table) in the program object modules for the extensions. This scheme can only be applied to shared libraries and dynamic link libraries that export a set of functions to the underlying executable. In this method, the export entries of these libraries are simply remapped to point to their extended counterpart. This means that an executable, importing functions off these libraries sees the extension in place. This method works at the binary level and can be static or dynamic depending on the export table implementation of the libraries under various OSs.

8.2.1.5 Branch Overwrite/Restore Technique

This is a very simple yet effective technique for extensions. Given below are the steps that come into play when this method is used:

1. Copy n bytes at the start of the target construct to be instrumented and store it in a buffer. Where n ¿= length of a unconditional branch construct for that particular machine architecture.
2. Insert branch construct at the start of the target construct, which jumps to the extended construct. For a majority of architectures, this will amount to just placing an unconditional branch instruction such as JMP.

3. During runtime, when the construct is invoked, the extended construct gets control that then performs the task at hand. If the extended construct needs to call the original unmodified target construct, it will perform steps 4 and 5.
4. Restore n bytes from the saved buffer to the memory location of the target construct and call the target construct to perform the default functionality.
5. Perform steps 1 and 2 again.

As seen, this method is fairly straightforward with the only exception of not being able to handle race conditions very smoothly. A race condition occurs when a call to the target construct occurs during the re-insertion of the branch construct. Due to this, such a call may not result in the extended construct getting control. What is worse is that, if the call occurs in while the copy is in progress, the system will be in an unstable state.

8.2.2 Deployment Mechanisms

8.2.2.1 Executable Image Modification

In this method, the executable file image is modified to include the payload. This is done by adding extra sections to the executable or by modifying existing sections to incorporate the payload. [9,12,15,16]. As seen, this is a static method that requires the knowledge of the executable image file structure in order to be able to incorporate the payload. Also since the payload is incorporated in binary, it has to be coded in position independent (delta code).

8.2.2.2 Executable Import Table Rewriting

This method of payloading is borrowed from the import table rewriting technique of extensions. However, in this case, the import table of the executable file is modified rather than the export table of the destination library. This technique injects the code

into the dynamic library list in the import table whereby the code is loaded automatically upon executable load [20]. Again, this is a static method requiring knowledge about the executable file image being manipulated.

8.2.2.3 Procedure Linkage Table Redirection

This method is very similar to the previous method in that Procedure Linkage Table (PLT) entries are replaced with exports from the shared library causing it to be loaded when the process starts. This scheme is applicable to executable image files that rely on the PLT and Global Offset Table (GOT) entries to resolve imports in shared libraries. An example is the Executable Link Format (ELF) found in most flavors of unices. This is also a static method.

8.2.2.4 Process Memory Space Injection

This technique employs finding free space in the process memory and then writing the Payload out to that and then deploys the extensions by hand. This method is dynamic but suffers from one serious drawback in that it fails if there is no free space in the process memory image to accommodate the code being injected. Also the injected code has to be coded in position independent code as the patch is applied in binary.

8.3 Overview of Sakthi

8.4 Extensions

Extensions are the mechanism by which desired portions of the host OS and supporting libraries can be changed to suit specific needs. We use the following terminology to describe the concept of Extension:

- Native Target Function (NTF) - This is the low-level construct that is to be extended. This construct could have calling conventions like stdcall, fastcall or cdecl or, it could be a pure assembly construct.
- Extended Target Function (ETF) - This is the user supplied function which does the following:
  - Performs any pre-processing pertaining to the new functionality.
  - Optionally calls the NTF through the Extender (see below).
  - Performs any post-processing pertaining to the new functionality.
  - Returns to the caller.

  The calling convention of the ITF must match that of the ETF. When instrumenting an arbitrary binary construct that is not a pure function, the ETF must be coded in such a way that it preserves the registers since we cannot rely on calling conventions. It might be argued that this is contrary to the architecture we propose, but recent compilers provide for functions to be coded in a naked style with inline assembly using, which, it is trivial to code ETF wrappers for constructs that are not pure functions.
- Extender - This code is responsible for replacing the NTF by the ETF and for providing a method by which the unmodified NTF can be successfully invoked (optionally) by the ETF.

8.4.1 Design

A construct can be abstracted as a function, expecting zero or more parameters and which does or does not return a value. Based on this fact let us consider a simple construct invocation shown in Figure 8.1a. Here a source program or module, at some point of its execution calls the NTF, which performs its duties and returns. A call here refers to an explicit branch instruction or simply the next instruction in sequence, in case

of instrumenting arbitrary constructs that are not pure functions. The example uses a hypothetical instruction sequence of a NTF in the IA-32 assembly language format (In reality, it could be in any machine architecture like the Sparc etc.). With extension in place, the construct invocation is as shown in Figure 8.1b. The ETF and the Extender are collectively called the Extension code.

Extension works on the concept of rewriting target functions in their inprocess binary image. In-process rewriting of target functions is a method by which the Extender is loaded into the target process space employing certain mechanisms (discussed later). The Extender then goes ahead and modifies NTFs, which are memory mapped in the process address space. This application of in-process rewriting is crucial to our design and gives our approach an elegant means to deploy extensions into a target process.

Extension consists of two steps. The Setup step modifies the NTF to enable transfer of control to the ETF. The Re-Insert step allows for calling the unmodified NTF if needed.

8.4.1.1 Setup step

Starting at the memory location of the NTF, enough instructions are disassembled so as to fit in a jump construct. These disassembled instructions are then stored in the Extender. This takes care of not straddling instruction boundaries thereby preserving complete instructions. A jump construct is then inserted at the NTF so that it transfers control to the ETF upon invocation. We note that, writing a jump construct comes with its own problems in case of multiprocessors, if it spans cache lines. However, as our emphasis is currently on single-processor systems, we will ignore this issue.

8.4.1.2 Re-Insert step

When the ETF wants to place a call to the original NTF, it does so through the Extender. The instructions that were disassembled originally in the setup step are

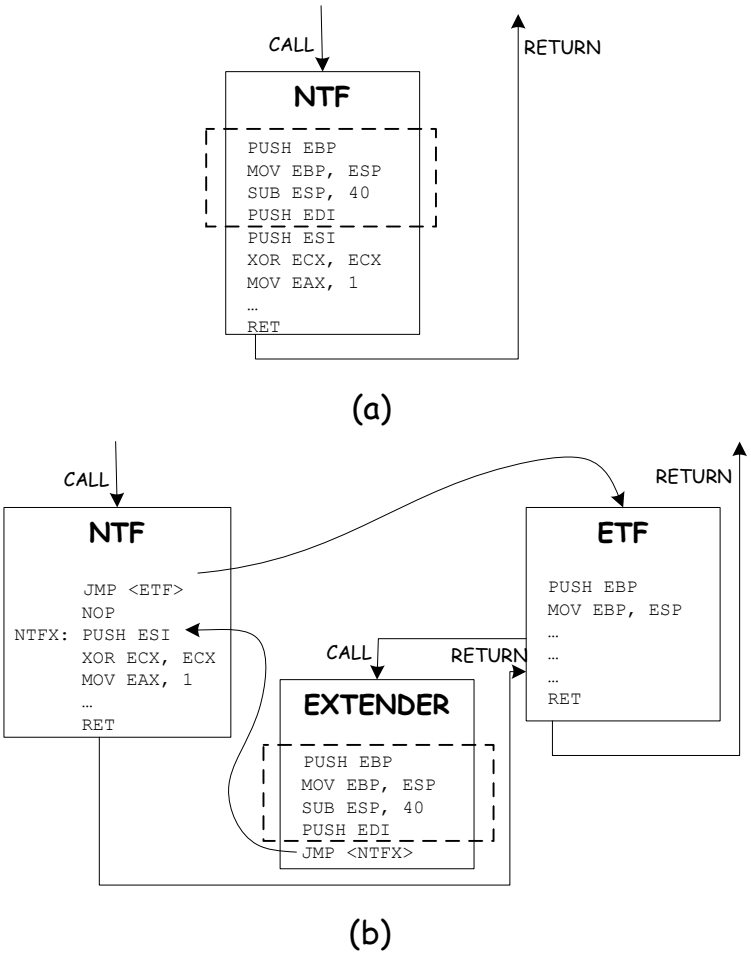


Figure 8.1. (a) Flow of call to NTF without extension, and (b) Flow of call to NTF with extension.

executed and then a jump is made to the following instruction (at NTFX) in the NTF. To the outside world this is logically equivalent to the model presented in Figure 8.1a, as the call is still made to the same NTF and the return from the ETF brings control back to the point from where the call was made as though the NTF had returned.

8.4.2 Implementation

Redirection involves machine specific code such as the insertion of the jump construct and the use of a partial disassembler. These vary for different machine architec-

tures. In most cases the jump construct directly corresponds to a jump instruction to the target location. However, this might not always be the case. For example, in the IA-32 architecture a jump instruction can refer to any arbitrary 32-bit target location, thereby encompassing the entire machine addressing space. Thus, for the IA-32 the insertion of the jump construct is basically insertion of the machine code for the long jump instruction followed by the target memory location (relative to the long jump instruction) in 32-bits as shown in figure 8.2a. In the Sparc architecture, there is no support for a single jump instruction to directly reference the entire addressing space. Thus, for the Sparc and other similar architectures, we need to make use of additional instructions as a workaround that allows us to jump to any target location in the machine address space as shown in figure 8.2b.

```
JMP <RELATIVE ADDRESS OF ETF>
```

(a)

```
SETHI %HI(ABSOLUTE ADDRESS OF ETF), %G5
OR %G5, %LO(ABSOLUTE ADDRESS OF ETF), %O7
JMPL %G0+%G5, %G0
NOP
```

(b)

Figure 8.2. (a) IA-32 Jump Construct, and (b) Sparc 32-bit Jump Construct.

We abstract machine specific code generated at runtime by isolating them in the instruction generator and the disassembler. Instruction generator is a module that is

responsible for generation of machine code for a specific instruction. It can be viewed as a partial assembler that generates machine codes for selected instructions. The disassembler is used by the extender to disassemble instructions at the start of the NTF, which are then copied to the extender code area as previously discussed.

8.5 Payloading

The extension code consisting of the ETF and its associated extender needs to be in the target address space for successful extension. Thus, the extension code must be loaded into the target address space. Here, target refers to either the kernel address space or a user-mode application or library address space. This process is called Payloading.

8.5.1 Payload

The extension code along with any associated support functions are collectively called the "Payload". A Payload in its simplest form consists of the following components:

- A one-time initialization procedure.
- A one-time un-initialization (reset or undo) procedure.
- The extension code for each NTF to be instrumented.

It is however possible to have other supporting code inside a payload but the abovementioned are the bare minimum requirements. Let us assume that there is a need to instrument k NTFs in the target address space. The payload will then contain k ETFs each with its extender as shown in Figure 8.3.

The initialization procedure is called only once, when the Payload is loaded in order for it to deploy extensions. This procedure makes use of the Setup step in the Extenders of the ETFs to deploy the instrumentation. The initialization procedure can also perform other Payload specific initialization such as opening RPC, sockets, IPC connections, message pipes etc.

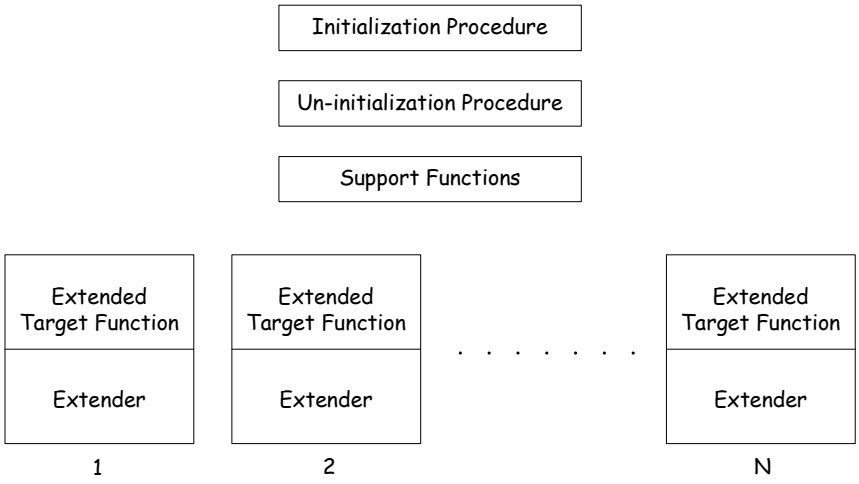


Figure 8.3. Components of a Payload.

The un-initialization or the reset procedure is called when the Payload is to be unloaded. This procedure restores the NTFs to their original state. It is also responsible for cleaning up Payload specific resources that were allocated. Thus, extensions are dynamic in that, one could load or unload a payload at will. Further, the framework includes the ability to selectively extend or restore NTFs at runtime.

8.5.2 Design

The payloading mechanism of Sakthi is dynamic and independent from the executable image file structure. The design of a unified Payloading scheme will have to take into account the goals of our framework - (1) to support global and local extensions (2) to be able to deploy extensions in newly created or already running processes and (3) to be able to implement the scheme on most OSs machine architectures. Accordingly, we present our Payloading design.

Each process has associated with it, two components. The first, its execution context which consists of the machine general purpose registers, floating point registers and other architecture dependent registers and the second its state (ready, running, suspended etc). The steps performed during Payloading are outlined below:

- Step-1: Spawns or attaches to Process-N. The process is now in the ready state.
- Step-2: Suspends Process-N.
- Step-3: Captures and saves the current execution context of Process-N (primary thread).
- Step-4: Creates a temporary code area in the address space of Process-N's primary thread (in implementation, this is mostly on the stack of the primary thread, but for some OSs, like most flavors of Unix, this might be the code segment) and generates code there too load the Payload:
  - call the initialization procedure of the Payload.
  - jump to the location in the saved execution context, thereby continuing the normal flow of execution.
- Step-5: Sets up the context of Process-N (primary thread) to start executing code created in the temporary area; Resumes Process- N.

8.5.3 Addressing Space Issues

In OSs that have a single execution space (DOS, VxWorks etc.) the Payload is a terminate and stay resident (TSR) executable or a shared library. Our Payloading scheme readily applies here albeit with all of the steps performing very little. Step-1 is performed only when a new process is to be launched. Attaching to a running process is trivial as all processes and the kernel share the same address space. Steps 2, 3, and 5 are void. During step-4 the framework simply loads the Payload as a TSR executable or a shared library.

In OSs with single-space execution global extension is the default, since all processes including the kernel share the same addressing space. Local extension is achieved by employing a filter on the executable load address and size or on unique process identifiers. That is, the process (es) to which the extension must apply are identified and when one of them is the current running process, the extension is enabled. In single address space OSs that lack process identifiers, one can trace through the memory control blocks (MCB) and find out the base address where the executable is loaded. Then, when the extended function is called, we can filter based on the return address on the stack by checking to see if it lies within the base and the limit of the MCB of the process executable.

In OSs that have multiple execution spaces, the Payload takes on various forms depending on the privilege level of the execution space. For kernel space, the Payload is simply a driver or a loadable kernel module. Most OSs provide a means by which a user-mode application can load and unload privileged code dynamically (device drivers or loadable kernel modules) into the kernel space for execution. The framework makes use of these services to load the Payload. Only Step-4 of the Payloading scheme applies and the framework loads the driver or the loadable kernel module during this step. OSs that do not provide a method, by which privileged code can be dynamically loaded and unloaded, provide a static method for the same. In such cases, the Payload must be listed in the list of statically loaded modules, which are automatically loaded into kernel space at system startup. However, all of the contemporary operating systems have a dynamic load and unload capability for privileged code.

Extension in kernel space is global by default. To achieve local extension, a filter on the unique process identifier is employed as described before. Payloading in user space is inherently more complex than in kernel space due to two possible implementations of user space, one with Copy-On-Write and the other without.

Copy-On-Write is conceptually similar to private copies of memory pages as far as writes are concerned. All the steps mentioned in the Payloading scheme are followed to deploy extensions. Extensions in Non-Copy-On-Write user space are global by default. In Copy-On- Write user space, they are local by default. Local extensions in Non-Copy-On-Write user space can be achieved as mentioned earlier by filtering on a unique process identifier.

However, implementing global extensions in Copy-On-Write user space is a bit cumbersome. This is done by enumerating all running processes in the system and loading the Payload into each one of them separately. We note that if performance is an issue then a kernel level global extension may be a feasible alternative. For example, consider a trivial application that needs to compress a stream of data that is sent over the network on the fly. This can be accomplished by using a global extension at the socket layer in user space or by global extension of the protocol driver in kernel space. However, if the function to be instrumented does not have a kernel level equivalent implementation (as in case of certain general purpose user level libraries), the strategy described here is the only alternative for global extensions.

8.5.4 Implementation

The dependency of Payloading on the underlying OS is greater than that of extension on the machine architecture. Payloading in a single address space OS is straightforward, since all memory is shared among processes and the kernel.

We present the outline of Payloading implementation for Windows and Linux, two popular OSs. However, it can be extended with ease to most OSs in wide use today (Solaris, VxWorks, WinCE, and other real time OSs). The steps outlined correspond to the OS specific implementation of the conceptual steps already described for Payloading in the previous section. A point to be noted is that, though the implementation steps

for the OSs might seem very specific, they are compatible among the same genre of OSs. For example, the implementation described for Linux is almost the same as for Solaris or any other flavor of Unix. We believe, that for most OSs our Payloading abstraction will almost always translate to a lightweight OS specific implementation.

8.5.4.1 Kernel-mode Payloading

Payloading in kernel space (in both Linux and Windows) is equivalent to that in single address space OSs, since the kernel is a single unit visible to all the processes at all times. Table 8.4 lists the Payload type and shows how step-1 and local extension is implemented in kernel space for various OSs. The Payload in each OS is wrapped with the necessary routines that conform to the OS generic device driver interface specification.

8.5.4.2 User-mode Payloading under the Windows OSs

For the most part, the user space organization of Windows 9x and ME OSs is very similar to that of NT, 2K and XP. However, Windows 9x and ME user space is non-Copy-On-Write whereas Windows NT, 2K and XP user space is Copy-On-Write. That is, under Windows 9x and ME, the system and shared DLLs are mapped at a shared memory (greater than linear address 0x80000000), which is visible to all user processes across writes. In case of NT, 2K and XP however, the system and shared DLLs are mapped privately in each process space.

The Payload in case of Windows 9x and ME is thus a Shared Dynamic Link Library, whereas in case of NT, 2K or XP it is a regular Dynamic Link Library. The Payloading steps are outlined below:

- Step-1: Call CreateProcess to create a new Win32 process. In case it is desired to attach to an existing process, the OpenProcess call is used instead.

Operating System	Payload Type	Payloading Steps	Local Extensions
Windows 9x and ME	Virtual Device Driver (VxD)	Step-1: use the CreateFile call to dynamically load the Payload (VxD).	This is achieved by employing a filter on the process identifier using VMM (Virtual Machine Manager) calls such as Get_Cur_VM_Handle and calls to VWIN32 VxD services.
Windows NT, 2K and XP	Kernel Mode Driver (KMD)	Step-1: register the driver using OpenSCManager and CreateService calls. Then run the driver using OpenService and StartService calls.	This is achieved by employing a filter on the process identifier using KMD support routines such as PsGetCurrentProcess.
Linux	Loadable Kernel Module (LKM)	Step-1: Use module support functions in the kernel to dynamically load the Payload (LKM).	This is achieved by employing a filter on the process identifier using system calls such as getpid which gives the identifier of the currently executing process in the kernel.

Figure 8.4. Kernel-Space Payloading Implementation for the Windows and Linux OSs.

- Step-2: Set the CREATE_SUSPENDED flag during a CreateProcess call or if attaching to an existing process, using the OpenProcess call and the OpenThread and SuspendThread calls subsequently. However, in case of Windows 9x and ME Win32 subsystem implementation, there is no in built function to obtain the primary thread handle. However, the original OpenProcess call under Windows 9x or ME actually has the capability to open threads in other processes. There is a check performed on the object handle to check whether it's a K32_EPROCESS or a K32_ETHREAD object. The call fails in case it encounters a K32_ETHREAD

object. The framework implementation includes a method using which the OpenProcess call is made to accept even thread objects by bypassing this check.

- Step-3: The GetThreadContext call is used for this purpose.
- Step-4: Write the temporary code on the process primary thread stack. Code for calling LoadLibrary to load the Payload is directly embedded into the primary thread stack.
- Step-5: Call SetThreadContext to restore the primary thread information and the ResumeThread call is used to re-start the process primary thread with the Payload loaded in its address space.

Global extension is the default behavior of the framework under Windows 9x or ME. This is due to the non-Copy-On-Write nature of the user space, making extensions visible to all the processes. To achieve global extension under Windows NT, 2K or XP we need to load the Payload in all the existing processes. First, we enumerate the running processes using the Process32First, Process32Next, Thread32First, Thread32Next and related calls. Now, for every process the Payloading steps discussed above are performed. Also, the CreateProcess call is extended in every process so that extensions can be deployed even on subsequent child processes that might be created.

Local extension is the default behavior of the framework under Windows NT, 2K or XP. This is due to the Copy-On-Write nature of user space. Hence, when the extenders are deployed, the extension is only visible in the process that initiated the deployment. Under Windows 9x and ME, local extension is achieved by employing a filter on the process identifier. The framework uses the GetCurrentProcessId call for this purpose. In every extender deployed, this call is made use to check whether the process for which extension is desired, is the one that invoked the construct that is extended.

8.5.4.3 User-mode Payloading under the Linux OS

Under the Linux OS, shared libraries wrap all OS system calls. Apart from these shared libraries there are other libraries that are process specific. The user space implementation is Copy-On-Write and the Payload is a shared library. The Payloading steps are outlined below:

- Step-1: Call fork, to create a new process. In case it is desired to attach to an existing process, the ptrace system call is used with PTRACE_ATTACH flag.
- Step-2: To suspend when attaching to an existing process is very simple with the ptrace system call as it is the operating semantic of the ptrace call. On a new process however, an infinite loop is inserted at the entry point and the process is made to continue its execution. By doing so we render it the status of an existing process thereby allowing us to apply the same treatment (use of ptrace) as already described. A point to be noted is that we cannot use the normal ptrace on the new process with PTRACE_TRACEME since we want to get control when the dynamic link loader (ld.so, ldlinux.so etc.) has been mapped into the target space.
- Step-3: Use the ptrace call with the GET_CONTEXT flag.
- Step-4: Most Unix systems make the stack non-executable, so is the case with Linux. The code at the current instruction pointer is first saved to a buffer. Then the code to load the Payload with dl_open call is embedded there in the code segment and finally terminated by an illegal instruction that causes an exception. Thus, the Payload loading code is executed and a fault is then reported to the framework. The framework then restores the code buffer and restarts the process at the previous instruction pointer, the only difference being that the Payload is now resident in the target process address space

- Step-5: Use the ptrace system call with the PTRACE_DETACH flag set, to let the process continue with the Payload loaded.

To achieve global extension under Linux we need to load the Payload in all the existing processes. First, we enumerate the running processes using the proc file-system interface. Then, for every active process, the Payloading steps discussed above are performed.

Also, the fork and the exec calls are redirected (using local extension, the default) in every process so that extensions can be deployed even on subsequent child processes that might be created. If a process calls fork to create another child process, then we have no problems as the child and parent share the exact copy of the address space that includes the extensions too. However, the situation becomes a bit complex if the child or parent replaces itself with an exec call. In this case, the address space is replaced and we need a mechanism by which we can deploy the extension. Issuing an implicit fork within the extended exec achieves this. When extended exec is called, it calls original fork to create a new child process. We have two parts after return from the call to original fork, the parent and the child. The parent part issues a call to original exec and never returns. The child waits for some time (to allow for the entry point loop of the executable to be executed), and then does a ptrace to attach to the parent process and insert the Payload into it. After that, the child terminates.

Local extension is the default behavior of the framework under Linux. This is again due to the Copy-On-Write implementation of user space.

8.6 Performance

The first part of this section discusses how Sakthi compares qualitatively to current approaches to extending the host OS. A quantitative evaluation in the later part sheds light on the overhead involved in applying our extension and payloading strategies.

Several alternative techniques exist for extending the host OS as already discussed in section 8.2. The most efficient among them is the target function rewriting technique. A detailed comparison of various schemes for extending the host OS and their performance is given in [20]. Figure 8.5 summarizes a qualitative comparison of our Payloading method with the ones discussed in section 8.2.

Payloading Technique	New Process	Running Process	Kernel	Payload format
Executable Image Modification	YES	NO	NO	PIC
Executable Import Table Rewriting	YES	NO	NO	Normal
Procedure Linkage Table Redirection	YES	NO	NO	Normal
Process Memory Space Injection	YES*	YES*	YES	PIC (Normal For Kernel)
Payloading in SAKTHI	YES	YES	YES	Normal

* Not always success
PIC = Position Independent Code (delta code)

Figure 8.5. Qualitative Comparison of Payloading under Sakthi with other strategies.

A quantitative evaluation of Sakthi involves assessing the contribution of two types of overhead (1) a one-time Payload load time and extender deployment time overhead and, (2) a runtime overhead. The measurements performed under Windows (98SE, ME, 2K) and Linux (kernel 2.4.0) running on a 1.5Mhz AMD Athlon-XP single-processor system, using user space payloading and local extension, are shown in Figure 8.6.

The Payload load time takes about 6–10 ms on an average for a new process and 5–7 ms on an average for an active process. Each extender deployment incurs an overhead of about 13ms on an average depending on the NTF and the underlying machine architecture. The runtime overhead is minimal and is only due to the jump construct that comes to an average of 3–4 us. The reader should note that these numbers give

Operating System	Payload Load time for new process	Payload load time for active process	Extender deploy time	Extender runtime overhead
Windows 98SE	6.230 ms	5.350 ms	11 μs	3 μs
Windows ME	7.113 ms	5.637 ms	14 μs	2 μs
Windows 2000	8.234 ms	7.223 ms	13 μs	5 μs
Windows XP	7.964 ms	6.858 ms	12 μs	4.2 μs
Linux	9.225 ms	7.533 ms	15 μs	3 μs

Figure 8.6. Overhead due to user-mode payloading and extensions.

an indication of what the equivalent values would be in other scenarios, i.e., user-mode global extensions and kernel-mode extensions.

8.7 Summary

In this chapter we have discussed Sakthi, a core subsystem of WiLDCAT that employs portable techniques that enable dynamic extension of the executing OS and the ability to inject and execute environment specific code-constructs in the context of the whole OS or a specified processes/threads.

CHAPTER 9

MISCELLANEOUS STEALTH TECHNIQUES AND NEXT GENERATION MALWARE SUPPORT

In this chapter we discuss miscellaneous stealth techniques that WiLDCAT employs to shield the environment from any form of detection or countermeasures. We also discuss how these techniques coupled with the design principles of WiLDCAT provide an environment conducive for various coding techniques employed by the next generation of malware.

9.1 Stealth Techniques

The code analysis strategies employed by WiLDCAT and the latency that they introduce can be detected during runtime albeit with some subtle tricks. We now present some techniques that the framework employs for stealthness to counter such methods.

9.1.1 Clock Patch

Cobra's SLE result in increased latency due to block creations and the xfer-stubs transferring control to and from the BCXE during block execution. Such latency is not present during normal execution of the target code-stream. Also VAMPiRE and SPiKE use the fault handlers (PFH and GPFH) for their implementation. The fault handlers also introduce latency is not present during normal execution.

A malware could use this fact to detect if its being analyzed. As an example, a malware could use the RDTSC instruction to obtain the amount of clock-cycles that have elapsed since system-bootup and obtain a relative measurement of its code fragment

execution. A malware could also use the real-time clock (RTC) to perform a similar latency detection in kernel-mode via the IN and the OUT I/O instructions.

Cobra's stealth implants prevent latency detection during SLE. The RDTSC instruction stores the return value in the EAX register. The RTC detection makes use of the RTC I/O ports using the privileged IN instruction. Cobra replaces the RDTSC instruction with a regular MOV instruction that stores the value of Cobra's internal processor-counter to the EAX register. Depending on the system VAMPiRE and SPiKE are run under, the handlers apply a clock patch resetting the time-stamp counter to a value that mimics the latency close to that of a normal handler. They also perform the same reset on the real-time clock since a malware could use the real-time clock to perform a similar latency detection.

9.1.2 Critical Structure Access

Malware running in kernel-mode are free to access any privileged structures. This can include page tables, page-directories, OS kernel regions etc. Since WiLDCAT uses virtual memory as a mechanism for code analysis, a malware running in kernel-mode could sweep through the page-tables and detect page attributes being non-present. A malware could also scan kernel-mode memory in order to detect WiLDCAT's core. The environment monitors all non-standard and standard code streams that are modified by using SLE and uses Cobra's feature of cloning to present the cloned copies of critical system structures to the malware thereby remaining completely stealth

9.1.3 Detection of SLE

Certain malware code streams can employ debugging techniques for their own execution. A malware for example can employ the POPF instruction to set the processor trap-flag. This results in a single-step exception being invoked. The malware can then use

the single-step handler to perform the actual functionality (eg. W32/Ratos). A stealth-implant in this case will replace the POPF instruction with a xfer-stub that tranfers control to the BCXE. The BCXE will then examine the trap-flag and will automatically generate a single-step exception for every instruction thereafter until the trap-flag is clear. Some code-streams running in kernel-mode can also employ the hardware debugging registers for computation (eg. W32/HIV and W32/Ratos). The debugging registers can also be used by the malware to set breakpoints within its code-streams. Cobra handles such issues by replacing access to such debug registers with stealth-implants and can generate breakpoint exceptions by monitoring the values in such registers.

Malware code-streams can also use instructions such as PUSH, VERW and ARPL in both user- and kernel-mode to obtain the selector for the executing code segment. Since Cobra executes the blocks at an unprivileged level such instructions will reflect an unprivileged code segment selector and can be used as a detection mechanism against the framework. However, Cobra's stealth-implants replace such instructions to reflect the actual value of the executing code segment selectors.

9.1.4 User-mode Framework Detection

A malware could also attempt to detect the environment via framework specific elements and signatures such as the instruments, ghosts and framework code/data in user-mode. In user-mode, the environment loads the instruments, ghosts and any framework code on demand in the target memory address space. When the environment specific code is finished processing, WiLDCAT ensures that any traces of itself are removed before returning control to the target code-stream, thereby preventing any form of detection.

A point to note for coarse-grained malware analysis using redirection is that: redirection involves modification to a target library, a malware cannot use this fact to establish a positive identification of the framework. This is because: (a) redirection preserves

the original instructions towards the start of the function and in many cases the complete function itself. It is not an easy task to detect a function split especially with the fact that functions in OS libraries undergo changes regularly in terms of service packs and dynamic updates. (as an example, service pack 3 on Windows 2000 added 5 new functions and changed around 50 different functions within NTOSKRNL.EXE). Further, the bridge instructions ensure that function split are not done using a single branch or milar instruction which can be detected, (b) the fillers added during redirection are such that the resulting file size closely resembles an updated component. The re-writing process will then allocate multiple functions per memory page if required in order to fit into the size, and (c) the rewriting is done so that the exports are maintained in the same order with relatively the same distance between them.

9.1.5 Single-Step Chaining

VAMPiRE's breakpoint persistence feature is used which employs a single-step handler. The single-step handler makes use of the host processor single-step exeption whose status is stored in a processor register. A malware could examine this value to detect single-stepping. VAMPiRE resolves such issues by maintaining a per-thread virtual single-step status and can simulate a single-step exception by chaining to the original handler in cases where the target malware uses single-stepping.

9.1.6 Multithread detection

Cobra's SLE leads to a couple of issues a malware could exploit to detect the framework during runtime. A malware in a multithreaded fashion can use a thread context capture function (under Windows OSs the GetThreadcontext API and under Linux, the ptrace API) to obtain the current program counter and stack contents for its executing threads. However, since the thread code-stream is being executed by Cobra, the values

of its program counter and stack pointer will be different than in the normal course of execution. Cobra instruments such APIs using our stealth coarse-grained instrumentation framework, SPiKE [38] and presents the original values of the thread program counter and stack pointer. A point to be noted is that Cobra has no effect on the thread stack of the target code-stream. The framework employs a local stack (different from the currently executing thread stack), that is switched to upon entry from an xfer-stub. Also, the xfer-stubs make use of the PIB to pass parameters to the BCXE, thereby ensuring that the thread stack is left untouched. This prevents Cobra from being detected using stack overflow mechanisms.

9.2 Support for Next Generation Malware

WiLDCAT, its architeture, its code analysis strategies and its stealth techniques equip the framework to tackle all the traits that embody not only the next but also future generation malware:

9.2.1 Self-Modifying

Self-modifying code support with coarse-grained analysis under WiLDCAT is ensured using breakpoint and instrumentation persistence which allow a target code stream to modify itself while still being able to trigger the breakpoint and instrumentation.

With fine-grained analysis, if a target code-stream modifies itself using its data and/or code segment, the environment handles it during block execution within SLE using the same mechanism as instrumentation persistence for coarse-grained analysis. For techniques that involve self-modification with program counter relative access, CTIs that employ the stack implicitly, are handled in a special fashion. As an example, on the IA-32 (and compatible) processors, the CALL instruction transfers control to a procedure unconditionally. The instruction pushes the return address on the stack as a part of its

semantic which is then popped by a corresponding RET instruction to resume execution at the caller. This property is exploited by many self-modifying code which, instead of using the RET instruction, pop the value into a register and use it to access their code in a position independent manner for modification. Cobra ensures that the program counter of the target code-stream is always reflected in the corresponding xfer-stub for such instructions thereby supporting self-modifying code.

9.2.2 Self-Checking

Since both coarse- and fine-grained code analysis strategies under WiLDCAT do not entail modification of the target code-stream, any self-checking code will therefore successfully execute under the environment. Also, as described in the case of self-modifying code, SLE ensures that the program counter of the target code-stream is always reflected in the corresponding xfer-stub thereby supporting any form of program counter relative self-checking methods as well.

9.2.3 Obfuscated Code

Coarse-grained analysis under WiLDCAT is performed by its core component SPiKE that uses stealth breakpoints to achieve instrumentation. Since stealth breakpoints make no change to the target code in any fashion, it supports any form of obfuscation automatically. Also, SLE that is employed for fine-grained analysis employs xfer-stubs for every CTI (unconditional or conditional). This adds support any form of code obfuscation, since obfuscated code rely on conditional and/or unconditional branches in between instructions for their functioning [61, 8, 14, 15]. Since every block generated by the BCXE terminates on exactly one CTI and the fact that the BCXE can handle both direct and indirect control transfers, it is guaranteed that the destination memory-

address for the next block creation always points to an address from which a valid block can be constructed using the disassembler.

9.2.4 Mutithreaded

Both coarse- and fine-grained code analysis techniques of WiLDCAT are completely reentrant and support multithreading. Thus any form of multithreaded code can easily be analyzed using the environment in both coarse- and fine-grained fashion. Further, with local and global instrumentation and selective isolation a malware analyst can choose a group of such multithreaded code streams for analysis while allowing the other code streams to execute as is.

9.2.5 Anti-analysis tricks

Coarse- and Fine-grained code analysis using WiLDCAT does not modify the target code-streams in any fashion and hence are completely invisible. The environment also employs miscellaneous stealth techniques to sheild itself from any form of detections or countermeasures as described in Section 9.1

9.2.6 Stealth Techniques

As described in Chapter 2, various next generation malware instrument portions of the host OS themselves and prevent other proceses from seeing critical resources allocated by the malware code-streams. However, with WiLDCAT it is very easy to monitor activities of such malware since the environment employs persistent instrumentation which allows the code-streams to deploy instrumentation over itself while still ensuring that it is the first to get control.]

9.2.7 Use of current analysis aids for functioning

The code analysis strategies employed by WiLDCAT do not use any form of processor supported analysis aids. Further, use of such aids within a code stream in both user and kernel-mode is intercepted by the environment which then simulates such aids. This allows a malware code stream to use these aids within their own code while ensuring that the environment has complete control over it.

WiLDCAT also allows a malware to employ the techniques employed by the environment itself. For example, a malware can install its own page-fault handler and implement breakpoints just like WiLDCAT for its own functioning. However, WiLDCAT will still be able to have complete control over such malware code-streams due to the technique of cloning which ensures that critical strucutres presented to non-standard code streams in both user and kernel-mode are not the original but a copy. Thus, in our example when a malware code-stream installs its own page-fault handler, it will do so in the copy of the critical structure responsible for exceptions (in this case the Interrupt Descriptor Table). Therefore, when a page-fault occurs the processor uses the original table which still points to the environment handler which then executes the malware handler using SLE in order to simulate a page-fault exception, while still having complete control over it.

9.2.8 Privileged and Unprivileged Execution

WiLDCAT's core resides in the highest privlege level within the host OS. This enables the environment to support coarse- and fine-grained analysis on both user- and kernel-mode code streams. Further, non-standard code-streams in kernel-mode always run in the supervision of the environment using SLE which ensures complete control over both user- and kernel-mode code streams at all times with the highest efficiency possible.

CHAPTER 10

EXPERIENCES

In this chapter, we discuss our experiences with several real-world malware such as W32/HIV, W32/MyDoom [44], W32/Ratos [74] and Troj/Feutel-S using WiLDCAT. These experiences are a result of numerous analysis sessions involving the malware and our prototype analysis tools WatchDog and ViPER. The analysis were carried out on a system running Windows XP and 98SE and an Intel 1.7 GHz processor with 512MB of RAM. The code fragments presented in this chapter are in the IA-32 Assembly Language Format.

10.1 W32/HIV

The `W32.HIV` is a dangerous per-process memory resident `Win32` subsystem virus that infects Windows executables and MSI (install) archives and corrupts certain important system files. The virus has a unique technique of *upgrading* itself from the internet and also possesses e-mail spreading abilities. Besides employing a stock encryption and polymorphic /metamorphic engine, the virus uses anti-debugging tricks to prevent itself from being analysed. It also has mechanisms of halting the machine completely if a debugger (e.g. Softice, Windbg etc.) is detected in use.

The `W32.HIV` virus and modified strains cannot be analysed using traditional software breakpoints as we describe in the following paragraphs. Even hardware breakpoints fail to help in the complete analysis of the virus. The first part of this section will present the analysis of the virus for code fragments of the virus where regular hardware breakpoints can be used. The second part of this section will then present an analysis of the

virus for code fragments where hardware breakpoints cannot be used thereby showing the utility of WiLDCAT.

For purposes of discussion, we will proceed to look at some simplified code fragments of the `W32.HIV` under different debugging sessions with our prototype debugger. We have removed details from the code fragments that are not pertinent to our discussion. Consider a code fragment as shown in Figure 10.1. This code fragment might not look interesting at a first glance. However, after quite a bit of tinkering, it is found that this, and other similar code fragments are in fact examples of the virus polymorphic/metamorphic engine kicking into action. More specifically, during our first encounter with debugging and tracing this code fragment, we had our suspicions on the section of the code fragment with the compare (`CMP`) instructions (lines 13–16, Figure 10.1), to have something to do with the main branch logic handling the functionality of the virus.

```
                ...
 1.  xor eax, esi
 2.  and eax, 38567fffh
 3.  add eax, ecx
 4.  cmp eax, edi
 5.  jbe 10015000
 6.  mov edi, eax
 7.  mov ecx, [esi+3ch]
 8.  xor ecx, [esi+30h]
 9.  mov [esi+3ch], ecx
10.  mov esi, [esi+40h]
11.  rep movsb
12.  mov eax, edi
Breakpoint ---->
13.  cmp eax, 5
14.  je 1001F0F0
15.  cmp eax, 10
16.  je 1001F1F0
17.  cmp eax, 4F
18.  je 1001F4F0
                ...
```

Figure 10.1. W32/HIV Self-Modifying Code Fragment.

However, when we set a traditional software breakpoint on the second `CMP` instruction (line 13, Figure 10.1) and let the code fragment execute, the breakpoint is not triggered. Tracing back a few instructions manually, we find the problem to be the section of the code fragment (lines 6–11, Figure 10.1), that generates new code (at runtime) overwriting existing code starting from the second `CMP` instruction (line 13, Figure 10.1).

This explains why the traditional software breakpoint was not triggered, as it was overwritten by the newly generated code. A manual workaround to this problem is achieved by tracing through the `REP MOVSB` instruction (line 11, Figure 10.1) one step at a time and inserting a traditional software breakpoint after the new code is generated. However, since a majority of the virus code is littered with such code fragments, this process soon becomes a tedious task.

Now, consider the code fragment as shown in Figure 10.2, which will be used to demonstrate the self-checking nature of the virus. The `W32.HIV` has self-checking abilities. This means that the virus has the capability to detect any kind of modification done to its code and will act in a way that will confuse the individual debugging the virus. The code fragment shown in Figure 10.2 is very well constructed to mislead someone who is trying to study the behaviour of the virus. Though, at a first glance it is very similar to the self-modifying code fragment as shown in Figure 10.1, it is just a plain sequence of instructions. In any case, when we set a traditional software breakpoint on the second `CMP` instruction (line 19, Figure 10.2) — to study the branching in detail — and let the code execute, the breakpoint triggers successfully. Everything appears normal, and there is no sign of any code changes etc. However, when we trace further, we arrive at a section of code that is garbage and leads to spurious faults. The problem is traced back to the section of the code fragment as shown in lines 11–18, Figure 10.2.

Here we see an integrity check being attempted on the code starting from the `CMP` instruction on line 19, Figure 10.2. Setting a traditional software breakpoint on this `CMP` instruction causes the instruction to be overwritten with a breakpoint instruction. However, this behaviour of traditional software breakpoints causes the code fragment to compute a wrong checksum during the integrity check (lines 11–18, Figure 10.2). Further, the virus is very intelligent in that, it does not terminate or branch to any location upon a failed integrity check. Instead, the checksum is the branch variable (stored in register

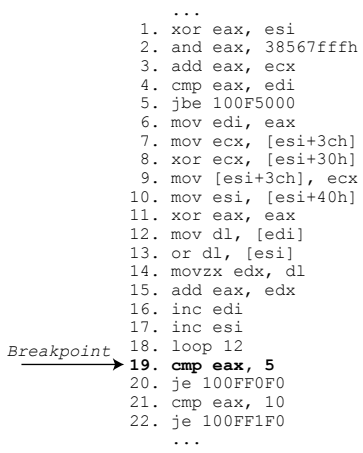


Figure 10.2. W32/HIV Self-Checking Code Fragment.

EAX) itself. In other words, the virus uses the checksum as a representative of the target address of a branch that performs some processing pertaining to the functionality of the virus. Thus, on an incorrect checksum, it branches to a location where the code is nothing but garbage and on a valid checksum the code fragment performs the function it was designed for.

The manual workaround in this case is achieved by precomputing the checksum and adjusting the value in the EAX register and tracing on. But, as with the self-modifying code fragments, there are many such self-checking code fragments. To further complicate matters, the self-checking code fragments are generated at runtime with predefined values for the branch variable (EAX) for each fragment. This predefined value is the value for which the fragment will perform the desired function. If, for a given self-checking code fragment, the branch variable (EAX) contains a different value, the functionality is undefined. This makes the process of precomputing and manually adjusting the value of the EAX register a tedious process.

As seen, traditional software breakpoints fail to help in analysing the virus because of its self-modifying and self-checking abilities. However, we can use ViPER under

WiLDCAT and traditional hardware breakpoints to set breakpoints on code fragments such as the ones shown in Figure 10.1 and Figure 10.2.

While it is possible to set hardware breakpoints on the code fragments discussed so far, we soon find that the virus has efficient anti-debugging capabilities that prevent the use of hardware breakpoints too.

Let us look at Figure 10.3 which shows a code fragment from the virus in the context of hardware breakpoints from a debugging session under the Windows 9x OS. The `W32.HIV` spawns (and eventually kills) several ring-0 (privileged) threads in the system at arbitrary intervals. The code used to spawn these threads are concealed within the virus polymorphic/metamorphic layers. The threads are different each time they are spawned, but have a common functionality as shown by the code fragment in Figure 10.3.

```
                ...
             1. sub esp, 8
             2. sidt [esp]
             3. mov eax, [esp]
             4. mov edx, [esp+2]
             5. mov eax, [edx+8]
             6. cmp eax, [esi+2bh]
             7. jb 11
             8. cmp eax, [esi+2dh]
             9. ja 11
            10. jmp 12
BADBEHAVE   ------>
            11. jmp CA50D000h
            12. mov dr0, edi
            13. mov eax, esi
            14. ror eax, 16
            15. mov dr1, eax
            16. mov ebx, dr2
                ...
```

Figure 10.3. W32/HIV Anti-Debugging Code Fragment.

The threads first check to see if the default single-step handler has been altered by checking the system interrupt descriptor table (IDT). It makes use of the `SIDT` instruction to grab hold of the single-step handler address. It then checks if the handler is pointing to something different from the expected system default range (which for Windows 9x is C0000000h–C000FFFFh). If the handler address is out of range, the thread knows that there is a debugger installed, and behaves in a very unfriendly manner (using the `JMP`

instruction) to throw the system into an unstable state as long as the debugger is active. This is shown in lines 1–11, Figure 10.3 (the actual behaviour is concealed behind the `JMP` instruction and is not shown here).

The second level of anti-debugging within the threads employ the debug registers themselves for computation and functioning as shown in lines 12–19, Figure 10.3. The only workaround to this second level, when using hardware breakpoints, is to manually compute the values in registers and trace them one at a time, while changing the instructions to hold registers other than debug registers. However, this is more than a mission, as (1) there are multiple threads one needs to change that are spawned and killed at regular intervals and (2) the threads themselves have self-checking code fragments, which means one has to set the instructions back to what they were before proceeding to trace further. However, using ViPER under WiLDCAT, it is possible to set breakpoints and trace through such code fragments without any problems.

10.2 W32/MyDoom

The `W32.MyDoom`, with variants commonly known as `W32.MyDoom.X-mm` (where X can be S, G, M etc.) is a multistage malware. It generally arrives via an executable e-mail. Once infected, the malware will gather e-mail addresses, and send out copies of itself using its own SMTP engine. It also downloads and installs backdoors and listens for commands. Once installed, the backdoor may disable antivirus and security software, and other services. The downloaded trojan might allow external users to relay mail through random ports, and to use the victim's machine as an HTTP proxy. The trojans downloaded by the `W32.MyDoom` envelope, generally posses the ability to uninstall or update themselves, and to download files.

The `W32.MyDoom` and modified strains cannot be completely analysed using current binary-instrumentation strategies. The malware employs a polymorphic code envelope

and employs efficient anti-instrumentation techniques to detect instrumentations and will remain dormant and/or create system instability in such cases. For the purposes of discussion, we look at a simplified code fragments of different variants of the `W32.MyDoom` under monitoring sessions with `WatchDog` under WiLDCAT. We have removed details from the code that are not pertinent to our discussion and have restructured the code for easy understanding. Consider a fragment of code shown in Figure 10.4a.

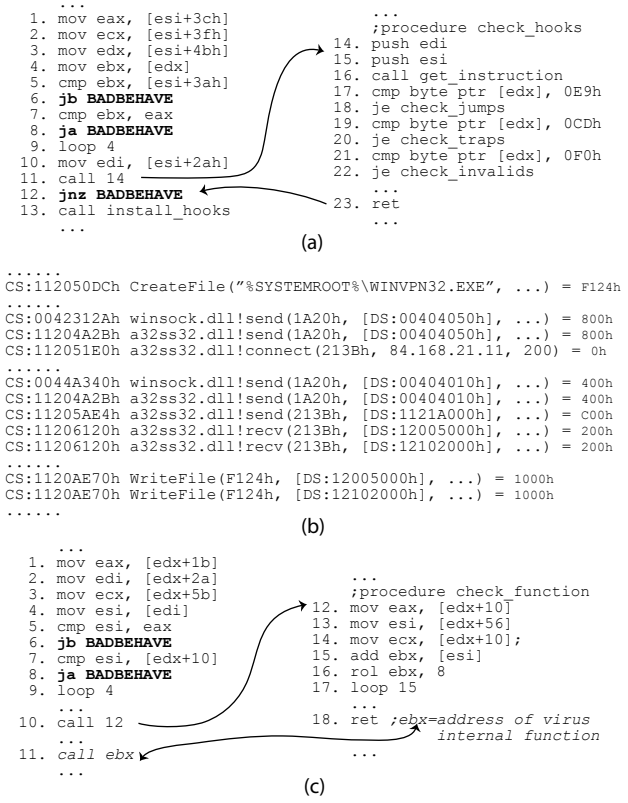


Figure 10.4. W32/MyDoom: (a) Instrumentation Check Code Fragment, (b) Behavior Log showing Trojan Download, and (c) Integrity Check Code Fragment.

The `W32.MyDoom` and its variants instrument (hook) several APIs within the system. One such API that is hooked is the `ZwQueryDirectoryFile`. The `ZwQueryDirectoryFile` API under the Windows OSs (NT, 2K and XP), is a kernel function that is used to ob-

tain the list of files within a particular directory. Calls to upper level file functions are ultimately translated to invoke this API within the kernel. The malware hooks this API, so that it modifies the return query buffer in a way that would prevent any regular application from seeing its critical files. There are several ways to instrument this API. The first method is by changing the pointers in the system service table pointed to by `KiSystemServiceDispatcherTable` to point to the instrumented functions. The pointer location for a particular system service can be found out by disassembling the dynamic library, `NTDLL.DLL` to get the indices for several system-calls. The second method is by using a probe-based instrumentation on the starting address of the API. Thus, if one instruments the API before-hand, using either of the two methods, it is easy to observe the behaviour of the malware, even if it hooks the API once again (since the malware will have to invoke the original API to obtain the populated query buffer in the first place).

However, the `W32.MyDoom` is intelligent to spot this. It performs a couple of checks to see if the API has been already hooked. The first check is a detection of a system-table hook (lines 1-9, Figure 10.4a). The malware uses a fact that original system-table entries point to code within `NTOSKRNL.EXE`, the Windows OS kernel. Thus, if one were to change the system-table entries to point to their own code, it will be at an address that is outside the image-space of `NTOSKRNL.EXE`. If this detection passes, the malware jumps to a location within its code which leads to a dormant operation or in certain cases causes system instability.

The second form of detection is for probe-based schemes at the target API. All probe-based schemes rely on the use of a unconditional transfer instruction (branch, trap etc.) at the address of the target code to be instrumented. This makes them an easy prey to detection. As seen from lines 14-22 of procedure `check_hooks` of Figure 10.4a, the malware checks to see if the code at the target address is any instruction that could potentially transfer control to an instrument (jump, trap, invalid instructions etc.). The

check sweeps through the target API address checking for such instructions before a conditional is encountered. Since current probe-based frameworks rely on the use of control transfer instructions at the start of the target function thats being instrumented, they are rendered unusable in this context. The idea of inserting the probe, after the malware has inserted its hook doesnt accomplish any functionality since it would never see the unmodified buffer, but is also defeated since there are several checks, scattered throughout the code of `W32.MyDoom`, within its polymorphic layers, to ensure that no other system can hook the APIs after it has.

However, using WatchDog it is trivial to instrument such APIs, while at the same time allowing the malware hook to be active. In effect, `WatchDog` bypasses the checks and can monitor such APIs in a stealth fashion. Figure 10.4b shows a sample log from the utility revealing a trojan download and implant.

Certain variants of the `W32.MyDoom` use localized DLL's for their operation. For example, the trojan renames `WINSOCK.DLL`, the dynamic library associated with socket functions, to a random name and replaces it with its own DLL instead. The replaced DLL is coded in a fashion employing polymorphic layers and incorporates malware functionality while finally chaining to the functions within the renamed `WINSOCK.DLL`. As seen from the log of Figure 10.4b, `WatchDog` captures both the replaced (`WINSOCK.DLL`) and the renamed (`A32SS32.DLL`) dynamic library invocations, allowing us to spot an activity such as a trojan download and implantation as shown.

The localized DLL's of `W32.MyDoom` employ certain anti-probing tricks. We noticed a couple of such tricks in one variant during our analysis. The code fragment of the malware in this context is shown in Figure 10.4c. The malware employs integrity checks using checksums on the socket primitives on the replaced `WINSOCK.DLL` (`check_function` procedure, Figure 10.4c). Thus, inserting traditional probes on such functions, results in erratic behavior (lines 6 and 8, Figure 10.4c). Solutions employing replacing the replaced

DLL or rehousing the EAT entries in the replaced `WINSOCK.DLL` are defeated by similar checksum fragments that are embedded within the malware code. If the malware detects a malformed replaced DLL, it will overwrite it with a new copy. As an added detection, the malware also checks for probe insertions in the renamed DLL (`A32SS32.DLL`) once loaded. Manual patching of such integrity checks are cumbersome since: (a) many such fragments are scattered throughout the malware code and (b) the checksum themselves are used as representatives of the target address of a `call` that performs some processing pertaining to the malware functionality (line 18 and 11, Figure 10.4c). Thus, on a valid checksum the `call` performs the desired internal function whereas on an incorrect checksum, it branches to a location where the code is nothing but garbage. With WatchDog under WiLCAT, instrumenting functions within the renamed DLL is trivial using redirection-pads. For instrumentation of the replaced DLL, drifters are inserted at desired functions directly.

10.3 W32/Ratos

W32/Ratos, with variants known as W32/Ratos.A, Backdoor.Ratos.A, Backdoor-.Nemog etc. is a trojan that runs under the Windows OSs. It is usually deployed as a second stage malware after being downloaded as a payload of other malware such as W32/MyDoom and its variants. W32/Ratos and its variants (hereon referred to as W32/Ratos collectively) execute in both user- and kernel-mode. Once in place, the trojan will allow external users to relay mail through random ports, and to use the victim's machine as an HTTP proxy. The trojan also has the ability to uninstall or update itself, and to download files by connecting to various predefined list of IP addresses and ports on various public file sharing networks.

The internal structure of W32/Ratos is as shown in Figure 10.5. The malware consists of a user-mode process (named DX32HHLP.EXE) and a kernel-mode compo-

nent (named DX32HHEC.SYS) running as a service. DX32HHLP.EXE is responsible for the bulk of the malware functionality with the kernel-mode component aiding encryption/decryption and initial deployment. W32/Ratos and its variants cannot be analyzed using current static approaches. The malware employs complex multithreaded metamorphic code envelopes for both its user- and kernel-mode components. W32/Ratos employs a multilevel encryption/decryption scheme employing algorithims which resemble the TEA [83] and IDEA [35] ciphers. It employs a windowed mechanism where only a small portion of it is decrypted in a pre-allocated region of memory, at a given time.

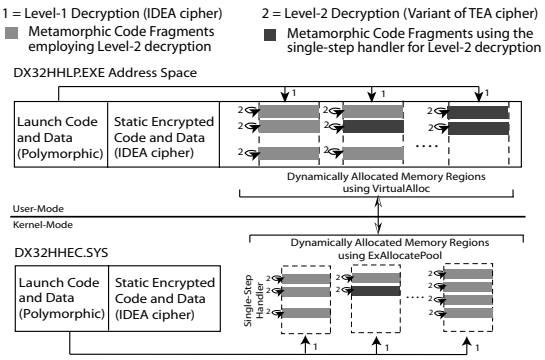


Figure 10.5. W32/Ratos Internal Structure.

The metamorphic code envelopes access code and data using relative pointers for self-modifications and integrity checks. Both the user- and the kernel-mode components employ several anti-analysis tricks to prevent themselves from being analyzed. Such checks include among other things, code execution time profiling and privilege level detections. These techniques are not handled by current dynamic fine-grained code analysis frameworks and the malware upon such detections will remain dormant or put the system into an unstable state. The trojan code envelop is multithreaded in both user- and kernel-mode and in certain cases achieve decryption via a single-step exception and also employ debugging registers for their internal computation, thereby defeating current de-

buggers. The kernel-mode component of the trojan achieves a stealth profile by hooking several Windows OS kernel functions such as ZwQueryDirInformation and ZwQwerySystemInformation.

The following paragraphs discuss in further detail, our experience in analyzing W32/Ratos, employing Cobra. The discussion serves as the basis on which we were able to document the inner structure (shown in Figure 10.5) and operation of the malware, thus illustrating the utility of the framework. For purposes of discussion, we will proceed to look at some simplified code fragments of W32/Ratos under different analysis sessions with WiLDCAT. The code fragments are shown in the 32-bit assembly language syntax of the IA-32 (and compatible) processors.

Our first step in analyzing W32/Ratos started with a coarse-grained analysis of the malware, thereby documenting its behaviour at a high level. We used WatchDog, to obtain the different system calls that were issued by the malware. We found that W32/Ratos issues several calls to ExAllocatePool, VirtualAlloc, VirtualProtect and MMProbeandLockPages, functions which allocate a range of memory outside of the existing code and/or data regions and change the attributes of the allocated memory range. The malware also creates several threads in both kernel- and user-mode as exempified by calls to PsCreateSystemThread and CreateThread APIs. We proceed to do a deeper investigation by using KiSwitchContext as our first overlay point and invoking Cobra for fine-grained analysis of the threads created by the malware. KiSwitchContext is an internal Windows OS kernel function that is used for thread pre-emption. We also instruct Cobra to generate events on memory accesses (reads, writes and/or executes) to any memory region allocated with the ExAllocatePool and VirtualAlloc APIs.

Consider the code fragment shown in Figure 10.6a. This (and several other) code fragments show the trojan metamorphic envelop in action. The code fragment shown here was obtained on events generated as a result of a write to the memory regions allocated by

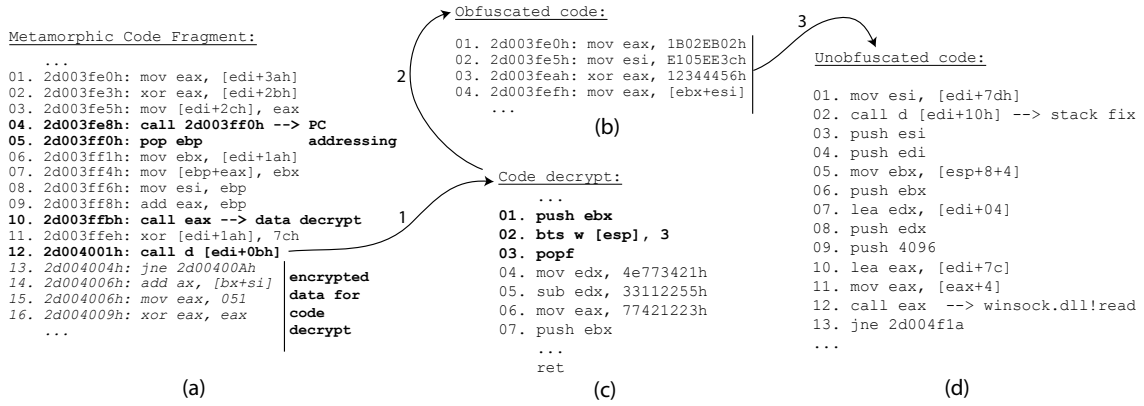


Figure 10.6. W32/Ratos Metamorphism: (a) Metamorphic Code Fragment, (b) Obfuscated Code, (c) Decrypted Code, and (d) Unobfuscated Code.

the malware using the ExAllocatePool and VirtualAlloc APIs and is a coalesced version of the blocks that were executed by Cobra during the analysis process.

W32/Ratos employs a multilevel decryption mechanism and metamorphic code engine to execute its internal code. The first level of decryption results in code fragments that are generated on the fly and executed in the memory regions allocated via the VirtualAlloc and ExAllocatePool APIs. Though the actual instructions of the decrypted code can vary from session to session of analysis due to the metamorphic nature, they have a regular pattern as shown in the example code fragment in Figure 10.6a. Here we see a fragment of a subroutine of the malware, that is responsible for its update feature.

Every code fragment generated by the metamorphic engine have two parts: a self-modifying second level decryption and an encrypted data that is responsible for the actual functioning of the fragment. The self-modifying section uses a program counter relative addressing (lines 4–5, Figure 10.6a) and modifies the code fragment by employing a second level decryption. Some of the metamorphic code fragments employ a second level decryption via a single-step handler in the kernel-mode component of the trojan. The second level of decryption changes the actual instructions at the start of the code

fragment and creates an obfuscated code section by employing the encrypted data of the metamorphic code fragment. It is also responsible for setting up the data for the new instructions that are decrypted. (lines 10 and 12, Figure 10.6a). The memory regions allocated via the VirtualAlloc and ExAllocatePool APIs thus act as a runtime window for code and data decryption on the fly. This technique ensures that not all of the malware code and/or data are in the decrypted form at a given time thereby making the analysis process harder. With Cobra however, it is relatively easy to document such techniques by using events corresponding to memory region accesses. Figure 10.5 shows the memory layout of W32/Ratos as a result.

The code generated by the second layer of decryption overwrites the start of the metamorphic code fragment and looks like a sequence regular instructions but are in fact obfuscated (Figure 10.6b). The actual instructions that are executed depend on the constants to the various instructions. Cobra's block execution events quickly reveal the actual code behind the obfuscated instructions as shown in Figure 10.6d. Upon further investigations it is found that the obfuscated instructions of the code fragment is actually a part of the trojan update feature that results in downlaods from certain public file sharing networks. A metamorphic code fragment in certain cases, can also include an integrity check on the code fragment to see if there has been tampering. In such cases, the trojan simply ensures that the second level decryption is voided, which results in the obfuscated fragment not being decrypted leading to spurious faults during the analysis process. Manual patching of such detections is a tedious process since the malware employs several such integrity checks within its code-streams. However, with Cobra, such integrity checks are not an issue since the framework ensures that the original code is left untouched in memory.

W32/Ratos employs a multilevel decryption scheme. The first level of decryption results in the generation of metamorphic code envelopes as discussed in the previous

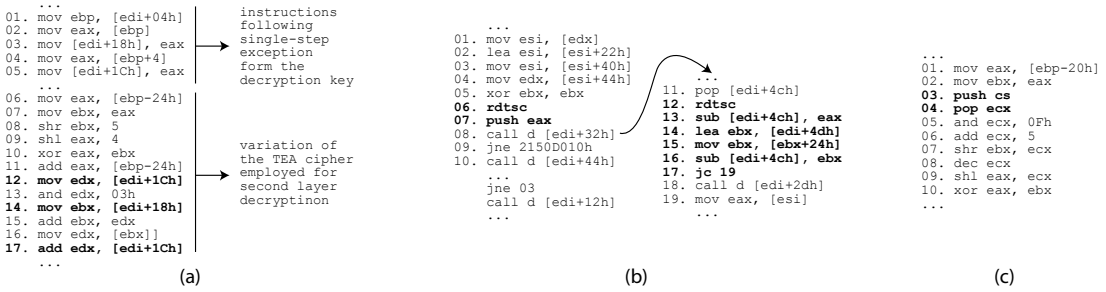


Figure 10.7. W32/Ratos Decryption Mechanism and Anti-analysis: (a) Single-step Handle, (b) Timing Check, and (c) Privilege Level Check.

section. The metamorphic code fragments employ a second level of decryption for their code and data. In certain cases the second level of decryption is performed using a sublte technique involving the single-step handler.

Consider line 12 of the code fragment shown in Figure 10.7a. This is responsible for the decryption of the code for the particular metamorphic code fragment. Now consider the code fragment shown in Figure 8c which is obtained when the call is executed by Cobra. As seen from lines 1–3, Figure 10.7c, the trojan uses the PUSHF instuction and sets the trap-flag thereby invoking the single-step handler from that point on, during execution. The single-step handler, present in the kernel-mode component and concealed within a polymorphic code envelope, is responsible for the actual second level decryption.

We proceed to analyze the single-step handler using Cobra, by setting an overlay point on the IDT entry for the single-step exception upon encountering block execution events containing the instructions shown in Figure 10.7c. We also setup Cobra for generating memory read and write events on the instructions shown in lines 4–7, Figure 10.7c and the encrypted data section (line 13 onwards, Figure 10.7a) of the metamorphic code fragment, since we suspected that that single-step handler might overwrite those with new instructions on the fly.

Figure 10.8a shows a part of the single-step handler that was reconstructed from the blocks by handling memory read and write events on the metamorphic code fragment in Figure 10.7a and the range of insructions shown in lines 4–7 of Figure 10.7c.

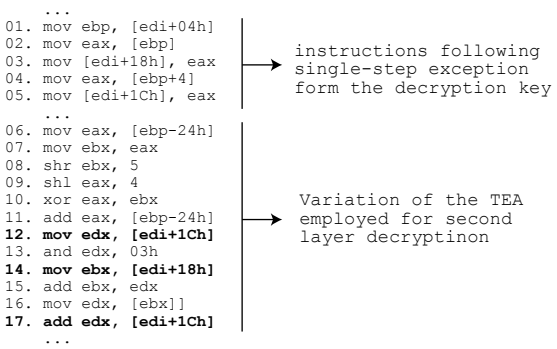


Figure 10.8. W32/Ratos Single-step Handler.

As seen from lines 1–5, Figure 10.8 (documented via a memory read event on the instructions shown in lines 4–7, Figure 10.7c), the single-step handler makes use of a 64-bit value following the single-step exception from the instruction sequence shown in Figure 10.7c. A memory write event on the encrypted data section of the metamorphic code fragment in Figure 10.7a, later reveals code fragments of the single-step handler (lines 6–17, Figure 10.8) which resembles a variant of the TEA encryption algorithm using a 64-bit key which is borrowed from the two instructions following the single-step exception (lines 5–6, Figure 10.7c). Thus, the encryption key is stored within the metamorphic code fragment itself, albeit camouflaged in instructions, which is used to decrypt the obfuscated section for that particular metamorphic code fragment. Similar investigations reveal the IDEA encryption algorithm being employed as a first layer of encryption.

W32/Ratos employs a couple of other techniques apart from its metamorphic code envelopes, to prevent its analysis. These techniques are scattered in various areas of

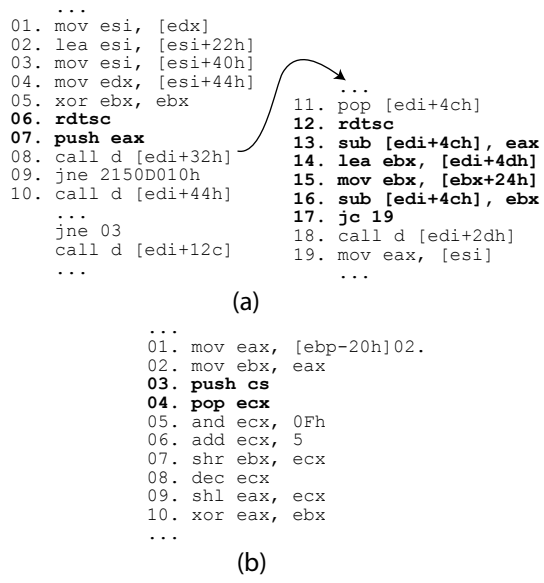


Figure 10.9. W32/Ratos Anti-Analysis Tricks: (a) Timing, and (b) Privilege Checking.

its internal code both in user- and kernel-mode. Consider the code fragment shown in Figure 10.9a. The malware employs a technique which we call code execution timing. The trojan makes use of the RDTSC instruction to obtain the value of the processor time stamp counter at various intervals within a group of instructions comprising its code (lines 6 and 12, Figure 10.9a). If the difference in the execution time of the instructions for a block is greater than a particular pre-calibrated threshold for that block of instructions (lines 13–17, Figure 10.9a), W32/Ratos sets certain specific flags which cause the decryption engine to cease working in a normal fashion (line 18, Figure 10.9a). Thus, when the trojan is executed in a virtual or supervised environment (resulting in an increase in the execution time beyond the the normal range) the metamorphic code generated by the trojan is erroneous and thus results in spurious faults.

Another technique employed by W32/Ratos is what we call privilege-level checks. The trojan obtains the privilege level of its currently executing code, data and/or stack segment during execution of instructions in kernel-mode. This is done using various

instructions that access segment selectors (PUSH, MOV etc.). Figure 10.9b shows a fragment of code employing the PUSH instruction to obtain the code segment privilege level within its decryption engine. The W32/Ratos employs a very intelligent mechanism in that it uses the privilege level indicator in the segment selectors as a part of its internal computations (lines 3–9, Figure 10.9b). Thus, if the code is run under a different privilege level to capture the behavior of the trojan in kernel-mode (this might be the case when running under a VM such as VMWare [81], Bochs [37] etc.), the decryption algorithm in this particular case would be erroneous as the values for the instructions would be completely different. However, Cobra handles such anti-analysis techniques with ease. The framework employs stealth-implants on blocks containing such instructions thereby fooling the malware into thinking its being executed without any form of supervision.

10.4 Troj/Feutel-S

Troj/Feutel-S is a malware that runs under the Windows OSs and belongs to the category of malware known as keyloggers. A keylogger is a program that runs in the background, recording all the keystrokes. Once keystrokes are logged, they are hidden in the machine for later retrieval, or shipped raw to the attacker, who then peruses them carefully in the hopes of either finding passwords, or possibly other useful information that could be used to compromise the system or be used in a social engineering attack. The Troj/Feutel-S is usually installed as a payload of rootkits and/or remote adminstration trojans such as the W32/MyDoom.

The malware is registered as a system service (usually named DriverCache) and includes a kernel mode component that installs its own system keyboard handler which is responsible for recording the keystrokes. The trojan also provides a proxy server that allows a remote intruder to gain access and control over the computer. It also employs a complex code envelope and contains a variety of anti-analysis tricks including hiding its

critical files, configuration and process information. The malware also has the ability to inject its code into other processes to provide application specific keylogging. Examples being Outlook Express (e-mails), Internet Explorer (web access).

Our first step in analyzing the Troj/Feutel-S to locate its internal key buffer involved behavior monitoring of the malware with respect to the system. We used Watchdog and found that the malware among other components loads a kernel-mode driver using the OpenSCManager API. We proceed to do a finer level investigation by first setting a hardware legacy I/O breakpoint on ports 60h and 64h (the 8042 keyboard controller ports) using our prototype debugger with the hunch that the malware would use them to capture key inputs in kernel-mode. However, we find that the entire system freezes or restarts when such a breakpoint is set. We then employ kernel-mode legacy I/O breakpoints using ViPER on ports 60h and 64h using the IRQ-1 exception handler and the KiSwitchContext as overlay/release points. The breakpoint in this case successfully triggers on the code stream within the IRQ-1 exception handler as shown in Figure 10.10a. We have removed details from the code that are not pertinent to our disccusion.

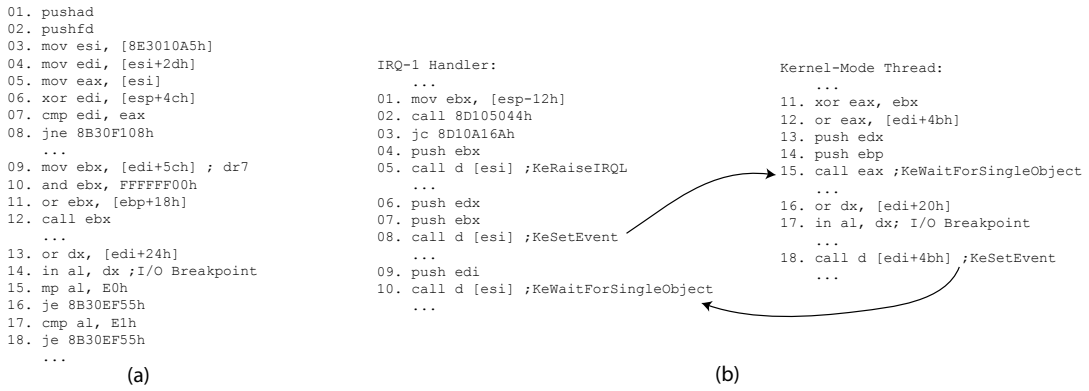


Figure 10.10. Troj/Feutel-S Keylogger: (a) IRQ-1 handler and (b) Kernel-Mode Thread I/O.

As seen from line 9, Figure 10.10a, the malware makes use of Dr7 debug control register (which is used to control hardware breakpoints) to check if any hardware breakpoints are set. However, the check is not as simple as just branching to a particular location on detection, but uses the bits of the Dr7 register to compute the target address of a branch (CALL instruction, line 12, Figure 10.10a). Thus, if there are any hardware breakpoints set, the target address for the CALL instruction is computed to be a spurious one and thus leads to a fault in kernel-mode which explains why the system freezes or restarts. A point to note is that the Dr7 contents cannot be easily patched manually since the location where its value is stored is variable, depending upon a specific instance of the polymorphic code envelope and such checks are found scattered throught the malware.

Troj/Feutel-S also employs I/O in its kernel-mode threads in certain cases instead of the IRQ-1 handler to further make it difficult to find the exact location where the keyboard I/O is done (Figure 10.10b). The kernel-mode thread waits on a kernel event that is set by the IRQ-1 handler (line 15, Figure 10.10b). When IRQ-1 triggers as a result of a keyboard activity, it signals the corrsponding event (lines 5–8, Figure 10.10b), upon which the kernel-mode thread performs the I/O involving the keyboard controller (line 17, Figure 10.10b). Once the I/O is complete, the kernel-mode thread signals the IRQ-1 handler, which then performs an end of exception returning to the kernel.

With kernel-mode legacy I/O breakpoints using ViPER, it is easy to set a legacy I/O breakpoint in both cases since WiLDCAT monitors the specified overlay/release points (in this case the malware threads and the IRQ-1 exception) revealing the area of code where the keyboard I/O is done.

10.5 Summary

In this chapter we have discussed our experiences with various malware such as W32/Ratos, W32/HIV, W32/MyDoom and Troj/Feutel-S which are representatives of

the next generation of malware. As seen from our experiences, current strategies in code stalling, coarse and fine-grained code analysis fail to operate in the realm of such next generation malicious code. However, using WiLDCAT and its components and tools we were able to analyze and document such malware from both a behavioral and structural perspective.

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

In this dissertation we have focused on various aspects of dynamic malware analysis including stealth stalling of executing code-streams, behavioral and structural analysis techniques and our experiences in analyzing various next generation malware using these techniques. Specifically, we have addressed the following problems and made the following contributions:

- Stealth Stalling of Code-streams: We have shown that existing stratgies in stalling executing code-streams are defeated in the context of next generation malware and have proposed radical techniques to stall execution of code-streams at runtime in a completely stealth fashion. Our stealth breakpoint framework codenamed VAMPiRE provides breakpoint ability for code,data and I/O in both user and kernel-mode on commodity OSs without any form of modification to the executing code-streams. We have carried out performance studies using our stealth breakpoint framework and have shown that the latency is well within limits to suit interactive analysis and in some cases even comparable to existing hardware and software based approaches of code stalling.
- Coarse-grained Malware Analysis: Malware Behaviour analysis is the first and critical step to obtain antidotes. First, we motivated the need for coarse-grained malware analysis in the context of several real-world malware. We identified the drawbacks of current approaches in coarse-grained code analysis and proposed a novel coarse-grained malware analysis framework to overcome those limitations. SPiKE, our coarse grained instrumentation framework provides the ability to mon-

itor and/or alter semantics of code constructs at runtime in a completely stealth fashion. The key feature of the framework is its persistent nature, in the sense that it allows an executing code stream to deploy its own instrumentation over itself, while still ensuring that it is the first to get control. We have comprehensively evaluated SPiKE by comparing it to existing instrumentation strategies both in a qualitative and quantitative fashion. Our qualitative comparison shows that the framework is teh first of its kind in tailoring an instrumnetation strategy for malware analysis while our quantitative comparsion shows that the framework is efficient enough for interactive monitoring and/or analysis.

- Fine-grained Malware Analysis: Malware Stuctural analysis is a critical step in obtaining a blueprint of the malware using which it becomes easy to tackle its future variants. First, we motivated the need for fine-grained malware analysis in the context of several next generation malware that are being coded in an iterative fashion. We identified the drawbacks of current approaches in fine-grained code analysis and have extended the current virtual machine technology to overcome those limitations. Cobra, our stealth-localized execution framework proposes radical techniques to construct an on demand, selective, localized and stealth execution environment in both user- and kernel-mode on commodity OSs. Cobra can analyze the execution of a code stream from as fine as an instruction level to a group of instructions. The framework can do so while allowing other code streams to execute as is. Cobra can execute privilege level code and allows manipulation of critical system structures while ensuring complete control using the mechanism of cloning. We have carried out performance studies using Cobra and have shown that the latency is within limits to suit interactive analysis
- Dynamic Extensions: The techniques proposed by us require modifications to certain parts of the host OS. We identified the drawbacks of current approaches and

proposed retargetable techniques to dynamically monitor and/or enhance parts of the host OS without access to sources as well as load portions of the environment in the context of a specific process/thread under the host OS. Our framework code-named Sakthi, enables binary instrumentation across multiple operating systems and machine architectures. It is based on the concept of dynamic extensions and provides a code injection mechanism, which lends itself to implementation on a wide range of commercial and free operating systems. Additionally our framework is capable of providing options for global and local extensions, is able to extend pure or arbitrary OS constructs and deploy code in a new or an already executing process within the OS with minimal downtime.

- Experiences: We have used WiLDCAT to analyze various malware such as W32/HIV, W32/Ratos, W32/MyDoom and Troj/Feutel-S which are examples of the next generation malware. These malware employ sophisticated techniques that thwart existing code analysis strategies and can be expected to become defacto in the near future if not more advanced. However, with WiLDCAT we were able to analyze these malware in a comprehensive fashion documenting both their behavior as well as structural organization thereby showing the impact and efficacy of our proposed techniques.

11.1 Future Work

We have shown how current code analysis techniques in the realm of malware have severe shortcomings in the context of next generation malware and have proposed novel techniques that enable analysis of not only the next, but also possible future variants. However, with the ever-evolving nature of malware, it is imperative that research involving strategies for malware analysis, always keep pace, if not be a step ahead, in order

to help malware analysts analyze and find antidotes in a timely fashion. The following sections discuss some interesting directions from here on for WiLDCAT.

11.1.1 Unobtrusive Execution

In certain cases, the analysis of a malware can put the system into an irrecoverable state. As an example, a malware that overwrites/deletes certain critical system files on a specified date might render the analysis system useless after just one analysis session on that particular date if the malware analyst executes code-streams that overwrites/deletes those components. While this is not a critical issue, since the malware analyst can choose to skip over the execution of such code-streams upon analysis, it is nevertheless desirable to place the onus of such decisions on the analysis environment rather than the analysts themselves. The notion of unobtrusive execution, is to prevent such situations by virtualizing critical OS subsytems such as files, network shares etc. This would allow a malware analyst to execute dangerous code-streams without having to worry about putting the system into an unstable state. Further, such virtualization must be completely undetectable.

11.1.2 Dynamic Reverse Analysis

Reverse analysis will allow a malware analyst to go back and forward in time during an analysis session in a dynamic fashion. This is a very useful feature if a malware analyst needs to re-analyze a particular code-stream. As an example, let us consider a scenario where a malware analyst is analyzing a malware to glean information about how it deploys itself and spreads. It would be convenient, if the analyst can go backwards and forwards in time in the locality of the deployment code within the malware, to study the deployment as many times as required instead of having to redeploy the malware every time. This is more useful in cases where various deployments might have different code

envelopes (metamorphism) so that the malware analyst would first have to locate the deployment code and then analyze it each time.

11.1.3 Distributed Malware Analysis

Many recent malware employ distributed attacks using a group of hosts. The time is not far when malware writers start to employ distributed code envelopes that scatter malware code modules among various infected hosts, rather than using a single payload with mutation as a vast majority of them do today. Distributed malware analysis would allow a malware analyst to deploy malware on a group of hosts and analyze individual components of the malware on various hosts concurrently, as they run in real-time. Also, network state and data are better virtualized for unobtrusive execution, employing a distributed analysis scheme than using a single-host.

REFERENCES

[1] Ashcraft, K., and Engler, D. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy* (Berkeley, California, May 2002), pp. 143–159.

[2] ATShield Ltd. Anti-trojan shield - high performance trojan detector. Available online at URL `http://www.atshield.com`. Last accessed 21 Feb. 2007.

[3] Beander, B. Vax debug: An interactive, symbolic, multilingual debugger. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* (1983).

[4] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[5] Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., and Tawbi, N. Static detection of malicious code in executable programs. *Symposium on Requirements Engineering for Information Security* (March 2001).

[6] Bergeron, J., Debbabi, M., Desharnais, J., Ktari, B., Salios, M., Tawbi, N., Charpentier, R., and Patry, M. Detection of malicious code in cots software: A short survey. *First International Software Assurance Certification Conference* (March 1999).

[7] Bergeron, J., Debbabi, M., Erhioui, M., and Ktari, B. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the IEEE 4th International Workshop on Enterprise Security* (June 1999).

[8] Bishop, M., and Dilger, M. Checking for race conditions in file accesses. *Computing Systems 9*, 2 (1996).

[9] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiezales, G., and Moon, D. A. Common lisp object system specification. *Sigplan Notices 23* (September 1988).

[10] Branco, R. R., and Lawless, T. The Saint Jude Project. *Sourceforge* (December 2005). Available online at URL `http://www.sourceforge.net/projects/stjude`. Last accessed 21 Feb. 2007.

[11] Bruening, D. Efficient, transparent, and comprehensive runtime code manipulation. *Ph.D. Thesis. Massachusetts Institute of Technology.* (2004).

[12] Buck, B. R., and Hollingsworth, J. An api for runtime code patching. *Journal of High Performance Computing Applications 14*, 4 (Winter 2000), 317–329.

[13] Cantrill, B. M., Shapiro, M. W., and Leventhal, A. H. Dynamic instrumentation of production systems. In *USENIX 2004 Annual Technical Conference* (Boston, MA, June 2004), pp. 15–28.

[14] Chen, H., and Wagner, D. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (November 2002), pp. 235–244.

[15] Chess, B. Improving computer security using extending static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (May 2002), pp. 160–173.

[16] Chess, D., and White, S. An undetectable computer virus. *Virus Bulletin Conference* (2000).

[17] Christodorescu, M., and Jha, S. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium* (Aug 2003), pp. 169–186.

[18] Christodorescu, M., and Jha, S. Testing malware detectors. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (July 2004), pp. 34–44.

[19] Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R. Semantic aware malware detection. In *2005 IEEE Symposium on Security and Privacy* (May 2005).

[20] Ciubotariu, M. Netsky: a conflict starter? *Virus Bulletin* (May 2004), 4–8.

[21] Cohen, F. Computer viruses: Theory and experiments. *Computers and Security 6* (1987), 22–35.

[22] Cohen, F. Operating system protection through program evolution. Available online at URL `http://all.net/books/IP/evolve.html`. Last Accessed: 22 February 2006.

[23] Collberg, C., and Thomborson, C. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering 28*, 8 (August 2002), 735–746.

[24] Collberg, C., Thomborson, C., and Low, D. A taxonomy of obfuscating transformations. *Technical Report 148, Department of Computer Science, University of Auckland* (July 1997).

[25] Compuware corp. Debugging blue screens. *Technical Paper* (September 1999).

[26] Debbabi, M., Girard, M., Poulin, L., Salois, M., and Tawbi, N. Dynamic monitoring of malicious activity in software systems. *Symposium on Requirements Engineering for Information Security* (March 2001).

[27] Giffin, J., Jha, S., and Miller, B. Detecting manipulated remote call streams. In *11th USENIX Security Symposium* (2002).

[28] Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium* (July 1996).

[29] Gordon, J. Lessons from virus developers: The beagle worm history through april 24, 2004. *Security Focus* (May 2004). Available online at

URL http://downloads.securityfocus.com/library/BeagleLessons.pdf. Last accessed 01 November 2005.

[30] Hunt, G., and Brubacher, D. Detours: Binary interception of win32 functions. In *3rd USENIX Windows NT Symposium* (Seattle, WA, July 1999), pp. 135–144.

[31] Intel corp. IA-32 intel architecture software developers manual, vols 1–3.

[32] Jensen, T., Metayer, D., and Thorn, T. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (May 1999).

[33] Kruegel, C., Robertson, W., Valeur, F., and Vigna, G. Static disassembly of obfuscated binaries. In *13th USENIX Security Symposium* (August 2004).

[34] Kruegel, C., Robertson, W., and Vigna, G. Detecting kernel-level rootkits through binary analysis. In *20th Annual Computer Security Applications Conference* (2004).

[35] Lai, X., and Massey, J. A proposal for a new block encryption standard. In *Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology* (1991), pp. 389–404.

[36] Larus, J., and Schnarr, E. Eel: Machine-independent executable editing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1995), pp. 291–300.

[37] Lawton, K. Bochs: The cross-platform IA-32 emulator. *Sourceforge* (January 2006). Available online at URL `http://bochs.sourceforge.net`. Last accessed 21 Feb. 2007.

[38] Linn, C., and Debray, S. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security* (October 2003).

[39] Lo, R., Levitt, K., and Olsson, R. Mcf: A malicious code filter. *Computers and Society 14*, 6 (1995), 541–566.

[40] Loukides, M., and Oram, A. Getting to know gdb. *Linux Journal* (1996).

[41] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwoo, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, June 2005), pp. 190–200.

[42] LURHQ. Sobig.e - evolution of the worm. *Technical Report. LURHQ* (2003). Available online at URL http://www.lurhq.com/sobig-e.html. Last accessed 01 November 2005.

[43] Maebe, J., R. M., and De Bosschere, K. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT02* (2002).

[44] McAfee. W32/mydoom@mm. *Virus Information Library* (2004). Available online at URL `http://vil-origin.nai.com/vil/`. Last accessed 22 Feb. 2006.

[45] McAfee Inc. The W9x.CIH virus. *Virus Information Library* (July 1998).

[46] McAfee Inc. The W32.HIV virus. *Virus Information Library* (October 2000).

[47] McGrath, R., and Akkerman, W. strace: system call tracer. *Sourceforge* (January 2007). Available online at URL `http://sourceforge.net/projects/strace`. Last accessed 21 Feb. 2007.

[48] Meushaw, R., and Simard, D. Nettop: Commercial technology in high assurance aplications. *VMWare* (2000). Available online at URL `http://www.vmware.com/pdf/TechTrendNotes.pdf`. Last accessed 21 Feb. 2007.

[49] Microsoft corp. Microsoft portable executable and common object file format specification.

[50] Microsoft corp. Windows defender. Available online at URL http://www.microsoft.com/athome/security/spyware/software/default.mspx. Last accessed 21 Feb. 2007.

[51] Moore, R. J. A universal dynamic trace for linux and other operating systems. In *FREENIX Track* (2001).

[52] Nethercote, N., and Seward, J. Valgrind: A program supervision framework. *3rd Workshop on Runtime Verification* (2003).

[53] Ogiso, T., Sakabe, Y., Soshi, M., and Miyaji, A. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals E86-A*, 1 (2003).

[54] Oney, W. Programming the microsoft windows driver model. *Microsoft Press* (December 2002).

[55] Paxon, V. Bro: A system for detecting network intruders in real-time. In *7th USENIX Security Symposium* (January 1998).

[56] Robbins, J. Debugging windows based applications using windbg. *Miscrosoft Systems Journal* (1999).

[57] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and B., C. Instrumentation and optimization of win32/intel executables using etch. In *USENIX Windows NT Workshop* (1997), pp. 1–7.

[58] Russinovich, M., and Cogswell, B. Procmon: Monitor process activity in real-time. *Sysinternals Freeware* (July 1997). Available online at URL http://www.sysinternals.com. Last accessed 21 Feb. 2007.

[59] Russinovich, M., and Cogswell, B. Portmon: Monitor i/o activity in real-time. *Sysinternals Freeware* (June 2000). Available online at URL http://www.sysinternals.com. Last accessed 21 Feb. 2007.

[60] Russinovich, M., and Cogswell, B. Tdimon: A general purpose network traffic monitor. *Sysinternals Freeware* (June 2000). Available online at URL `http://www.sysinternals.com`. Last accessed 21 Feb. 2007.

[61] Russinovich, M., and Cogswell, B. Filemon: Monitor filesystem activity in real-time. *Sysinternals* (August 2005). Available online at URL `http://www.sysinternals.com`. Last accessed 22 Feb. 2006.

[62] Russinovich, M., and Cogswell, B. Regmon: Monitor registry activity in real-time. *Sysinternals* (August 2005). Available online at URL `http://www.sysinternals.com`. Last accessed 22 Feb. 2006.

[63] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., and Soffa, M. Reconfigurable and retargetable software dynamic translation. In *1st Conference on Code Generation and Optimization* (2003), pp. 36–47.

[64] Singh, P., and Lakhotia, A. Analysis and detection of computer viruses and worms: An annotated bibliography. *ACM SIGPLAN Notices 37*, 2 (February 2002), 29–35.

[65] Sophos. Troj/feutel-s. *Virus Analyses* (October 2005). Available online at URL `http://www.sophos.com/virusinfo/analyses/trojfeutels.html`. Last accessed 05 Jan. 2006.

[66] Spitzner, L. Honeypots: Tracking hackers. *Addison-Wesley* (2003). ISBN: 0-321-10895-7.

[67] Srivastava, A., Edwards, A., and Vo, H. Vulcan: Binary transformation in a distributed environment. In *Technical Report MSR-TR-2001-50, Microsoft Research* (2001).

[68] Srivastava, A., and Eustace, A. Atom: A system for building customized program analysis tools. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 196–205.

[69] Symantec. Understanding and managing polymorphic viruses. Available online at URL `http://www.symantec.com/avcenter/whitepapers.html`. Last Accessed: 22 Feb 2006.

[70] Symantec. Backdoor.dkangel. *Symantec Security Response* (May 2002). Available online at URL `http://securityresponse.symantec.com/avcenter/`. Last accessed 20 Mar. 2006.

[71] Symantec. W32.lamin.b. *Symantec Security Response* (Nov 2003). Available online at URL `http://securityresponse.symantec.com/avcenter/`. Last accessed 20 Mar. 2006.

[72] Symantec. W32.kipis.k@mm. *Symantec Security Response* (Feb 2005). Available online at URL `http://securityresponse.symantec.com/avcenter/`. Last accessed 20 Mar. 2006.

[73] Szor, P. The art of computer virus research and defense. *Addison Wesley in collaboration with Symantec Press* (2005).

[74] TrendMicro. Bkdr.surila.g (w32/ratos). *Virus Encyclopedia* (August 2004). Available online at URL `http://www.trendmicro.com/vinfo/virusencyclo/`. Last accessed 22 Feb. 2006.

[75] Unix System Laboratories. Tool interface standards. elf: Executable and linkable format.

[76] Vasudevan, A., and Yerraballi, R. Dynamic malware analysis using stealth binary-instrumentation. *ACM Transactions in Information and System Security*. Submitted for review on 03/02/2007.

[77] Vasudevan, A., and Yerraballi, R. Sakthi: A retargetable dynamic framework for binary instrumentation. In *Hawaii International Conference in Computer Science* (Honolulu, HI, January 2004), pp. 652–662. ISSN:1545-6722.

[78] Vasudevan, A., and Yerraballi, R. Stealth breakpoints. In *21st Annual Computer Security and Applications Conference* (Tucson, AZ, December 2005), pp. 381–392.

[79] Vasudevan, A., and Yerraballi, R. Cobra: Fine-grained malware analysis using stealth localized-executions. In *27th IEEE Symposium on Security and Privacy* (Oakland, CA, May 2006).

[80] Vasudevan, A., and Yerraballi, R. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *29th Australasian Conference in Computer Science* (Tasmania, Hobart, January 2006), pp. 311–320.

[81] VMWare Inc. Accelerate software development, testing and deployment with the vmware virtualization platform. *Technical Report, VMWare Technology Network* (June 2005).

[82] Wang, C., Davidson, J., Hill, J., and Knight, J. Protection of software-based survivability mechanisms. In *Proceedings of International Conference of Dependable Systems and Networks* (2001).

[83] Wheeler, D., and Needham, R. Tea, a tiny encryption algorithm. In *Proceedings of the 2nd International Workshop on Fast Software Encryption* (1995), pp. 97–110.

[84] Wroblewski, G. General method of program code obfuscation. In *International Conference on Software Engineering Research and Practice* (June 2002).

[85] Yaghmour, K., and Dagenais, M. R. Measuring and characterizing system behavior using kernel-level event logging. In *USENIX Annual Technical Conference* (2000), pp. 13–26.

[86] Yetiser, T. Polymorphic viruses, implementation, detection and protection. *VDS Advanced Research Group, P.O. Box 9393, Baltimore, MD 21228, USA*. Avail-

able online at URL `http://vx.netlux.org/lib/ayt01.html`. Last accessed 22 Feb. 2006.

[87] Zhang, X., Wang, Z., Gloy, N., Chen, J. B., and Smith, M. D. System support for automatic profiling and optimization. In *16th Symposium on Operating System Principles* (1997), pp. 15–26.

BIOGRAPHICAL STATEMENT

Amit Vasudevan was born in Calcutta, India in 1979. He received his B.E degree from Bangalore University, India, in 2001, his M.S and Ph.D. degrees from the University of Texas at Arlington in 2003 and 2007, respectively, all in Computer Science and Engineering. From 2003 to 2007, he was with the department of Computer Science and Engineering, University of Texas at Arlington as an Assistant Instructor. His current research interests are in the area of Computer Systems Security and Operating Systems. He is a member of several IEEE and ACM societies and special interest groups.