# Covert Debugging
# Circumventing Software Armoring Techniques

Danny Quist
Valsmith
Offensive Computing, LLC
{dquist,valsmith}@offensivecomputing.net

## ABSTRACT

Software armoring techniques have increasingly created problems for reverse engineers and software security analysts. As protections such as packers, run-time obfuscators, virtual machine and debugger detectors become common, newer methods must be developed to cope with them. In this paper we will present our covert debugging platform named Saffron. Saffron is based upon dynamic instrumentation techniques as well as a newly developed page fault assisted debugger. We show that the combination of these two techniques is effective in removing armoring from most software armoring systems.

## 1. INTRODUCTION

Software programs are becoming more difficult to reverse engineer and analyze. A variety of methods are being used to prevent standard disassembly techniques. These methods were pioneered in the realm of anti-piracy and intellectual property protection but have recently found their way into malicious software for the purposes of preventing analysis and defense. These methods are often called obfuscation, packing or armoring.

There are many ways that software can protect, or armor itself, from analysis. The first is to perform simple debugger detection. On Windows systems this is done by analyzing the process execution block (PEB) for the presence of the debugger bit. Methods exist for toggling this bit, while still retaining the debugging functionality. Unfortunately anti-debugging methods have compensated for this. Methods such as int3 scanning, which look for the presence of a debugging instruction call, are effective at circumventing detecting debugger access. Hardware debugging detection and memory debugging methodologies also exist [1] for determining a non-standard system arrangement.

A common method for instrumenting application behavior is to use a virtual machine to simulate a full running environment. This has the benefit of isolating the running code inside of a self-contained environment which can be more closely controlled than raw hardware. There are several software armorning techniques that can be used for generically detecting the presence of virtual machines. All current virtual machines exhibit identifiable characteristics which can be used to change program operation. [2]

One of the more insidious and difficult to analyze forms of binary obfuscation is the shifting decode frame. This partially decodes a running program, executes that code, and then re-encodes it before executing a new portion. This provides the greatest difficulty for decoding, disassembling and debugging. [3]

Software armoring is becoming heavily used by malware. Legitimate software has been using techniques like these to protect themselves from analysis and modification for some time. Windows Server 2003 and Windows Vista employ a system to protect their internal workings. [4,5] Other software is using these systems to both reduce the size of their distribution and prevent reverse engineering. This presents a great difficulty to the security analyst for both understanding and assessing risk of applications as well as analyzing and defending against malware threats.

This paper will present two methods for generically removing packer and obfuscation from executable binaries. The first technique will show that using dynamic translation can be useful in finding insight into the post-decode, non shifting-decode-frame method of software armoring. The second method is to modify the pagefault handler of an operating system to monitor code execution. To make sure the pagefault handler is fully involved in the entire process, page-marking techniques similar to that of OllyBonE [6], PaX [7], and Shadow Walker [8] will be employed. Using these two methods it will be shown that effective single-stepping methods can be employed against common packing and software packing methods used by malware.

The organization of the paper is as follows. An in-depth discussion of the methods and techniques for detecting debuggers will be presented as well as common circumvention methods. Next virtual machine detection methods will be shown. Next the requirements for a covert debugging framework will be presented. Dynamic translation will be discussed as a solution. OS assisted methods, centering on the page-fault handler will be presented in detail. The application of the techniques will be shown with regard to malware analysis. Finally fundamental issues in detection of this mechanism will be presented, as well as future steps to prevent discovery and preserve access to the debugging system.

### 1.1 Related Work

Dynamic instrumentation has been used to monitor program processes previously. The method of automatically tracing program execution for the purpose of extracting relevant information has been proposed for the SPiKE framework [9]. Runtime debugging of malware has also been proposed by

Cifuentes, Waddington, and Van Emmerik as a method for rapidly understanding program execution. [10] The PIN system has been used by Ma, Dunagon, et. al for tracing the injection of malicious code into a vulnerable service. [11] The importance of analyzing dynamic behavior was also illustrated by the TTanalyze tool. [12]

The memory management process creates several opportunities for subversion. Offensive-uses of this system include the rootkit Shadow Walker used this method to reroute memory references rootkit discovery prevention. Shadow Walker is used to hide rootkit memory pages from the operating system or non-shadow walker executables. It does this by marking all its pages as non-paged, and then marks the dirty-bit. This causes the pagefault handler to be invoked for every memory access to the root kits memory pages. Using a subverted memory paging system, the rootkit points non-rootkit accesses to gibberish sections of memory. When it receives paging requests it allows them through. [8]

The PaX system uses the page fault mechanism to implement no-execute on systems without hardware support for the Linux operating system. PaX marks all memory pages with the supervisor flag. This causes the hardware pagefault handler to invoke the page-fault trap handler to accommodate the error. PaX simply determines whether an attempted execute was the result of an execute operation. If the result is an execute operation, then a fault is raised and the program is terminated. [7]

OllyBonE uses a similar techinique as PaX and Shadow Walker to implement break-on-execute. It interfaces with OllyDbg to be able to break on regions of memory. This allows the reverse engineer to identify where a possible execution region for the original entry point is at, and then trigger a debug at that point. OllyBonE uses the supervisor bit of the page table as well. [6]

## 2. SOFTWARE ARMORING TECHNIQUES

It is useful to have an understanding of the methods used by reverse engineers to gain an understanding of how to stop them. Reverse engineers have a common set of tools that are used to find useful information from a binary. The goal of the developer protecting her code is to prevent the reverse engineer from discovering how it works. In this context it is useful to analyze the techniques from both sides of the conflict. This section will discuss packing, virtual machine detection, debugger detection, and finally the shifting decode problem.

### 2.1 Packing / Encryption

Packing is the method that an executable uses to obfuscate an executable or to reduce its size. Packers are typically implemented with a small decoder stub which is used to unpack, or deobfuscate the binary in question. Once the decoding or "unpacking" process is complete, the decoder stub then transfers control back to the original code of the program. Execution then proceeds similarly to that of a normal executable. Packers create problems for malware analysts. First, current methods that are generically available require the analyst to manually single-step a debugger in order to find and expose the actual code executable or to analyze the assembly of the decoding stub in-depth in order to write a decoder. Second, many techniques such as those described in section 2.3 can be used to thwart the unpacking process.

### 2.2 Virtual Machine Detection

Detecting the presence of a virtual machine is one of the most important methods available to the malware author to protect his code from analysis. Typically under normal operations a malicious program will never be run inside of a virtual machine. The typical targets for malware authors are stand-alone systems that are being used for everyday tasks such as email, and online banking. If a virtual machine is detected, it is most likely being used to analyze the malware. Due to inherent flaws in the X86 architecture, virtualization cannot be supported at the hardware level. Certain instructions are known to be problematic. [13] As such there are a few common methods that are available to detect these.

The common theme throughout all of the advanced virtual machine detection methodologies is a single instruction that must yield the same results in ring-0, or the kernel execution space, and at the user privilege execution space. For the x86 architecture these are composed of the SLDT, SIDT, and SGDT instructions. The malware author can simply perform these instructions, and compare the results afterwards. Results will be different for virtual machines executing these instructions when compared to real-hardware executing them. [2] One method that can be used to circumvent this detection is to disable "acceleration" inside of a virtual machine environment. (in this case VMWare) This degrades performance but is usually sufficient enough to evade detection. Unfortunately when running in the non-accelerated mode there are still processor implementation discrepancies that can be used to identify the presence of a virtual machine. [14]

### 2.3 Debugger Detection

Debugging a running executable is one of the most powerful techniques available to a reverse engineer to quickly understand program execution. Using this method, it is possible to see the actual run-time dynamics of an executable, as well as monitor system calls. Unfortunately the presence of a debugger is trival for the process being debugged to detect. This section will discuss process debugging in detail.

#### *2.3.1 Windows Debugging API*

The Windows operating system implements a robust API for developing custom debuggers for applications. It is implemented using a call-back method which allows the operating system to single-step a running program at the machine instruction level. This API is used by the OllyDbg, WinDbg, and Visual Studio debuggers. The API allows the running program to receive events based on pre-set instruction level flagging. Detection of this type of debugger is as simple as looking at the process execution block PEB for a running program. The PEB is a data-structure that contains information relevant to the running program inside of the

Windows operating system. One field that is available inside the data-structure is BeingDebugged field. If this bit is set, it indicates that a debugger is attached to the process. Fortunately for the analyst, this bit can be toggled without losing the debugging capability.

### 2.3.2 INT3 Instruction Scanning

The next method used to implement a debugger is the INT3 instruction, sometimes referred to as a breakpoint exception. This instruction causes a CPU trap to occur in the operating system. The trap is then propagated to the running program via the operating system and passed to the running program. This provides a method by which a developer can set a breakpoint. However, programmers almost never put int 3 instructions directly into their programs so it is likely that if this is observed, a process is being monitored. Malware authors have implemented various methods to scan for the presence of this INT3 instruction, and alter execution if it is found. A simple CRC check or MD5SUM can detect and validate that the code has not been altered by an INT3.

### 2.3.3 Unhandled Structured Exception Handlers

Structured exception handler (SEH) unpacking creates another interesting problem for the reverse engineer. SEH are a method of catching exceptions from running applications. These are used when a particular program has a runtime error. Normally when an SEH is reached execution is passed to the handler the program developer has defined, or treated as an unhandled exception and execution halts. Malware authors have seized this as a method for implementing an unpacker. The malware author inserts a SEH and their own handler. This handler is typically a set of unpacking instructions. The SEH frame contains a pointer to the previous SEH frame and a pointer to the exception handler for the current frame. By triggering SEH exceptions the stack of a malware program is unwound until an appropriate handler is found. Due to the nature of the debugging interface, the debugger will insert its own SEH handling onto this stack. When the debugged program is run, it will raise an exception. This causes the debugger's stack to catch and handle the SEH instead, possibly crashing the debugger and preventing the malware from unpacking itself. Since there is no way for the debugger to discern between an exception generated by an error in its program, and the debugged program, this typically thwarts unpacking. Debugging programs such as OllyDbg have implemented methods to allow the reverse engineer to either handle the exception inside the debugger, or hand it to the debugged application's stack. This can be a very manual and tedious process if many SEHs are used.

### 2.3.4 Mid-Instruction Jumping

Typically a debugger will try to interpret the machine code of a running executable and print out more human readable output. Given the non-fixed-size of the Intel instruction set, this creates many opportunities for obfuscation of the run-time execution. A typical trick that can be performed is to take a long instruction and the value 0x90 as a parameter. This last parameter, interpreted on its own is the nop, or no-operation instruction. This will cause the CPU to run to the next instruction and continue execution.

### 2.3.5 Shifting Decode Frame

Shifting decode frame is a method by which a portion of the executable is unpacked, executed, then re-encoded. This method has the effect of preventing static post-execution analysis. This precludes the ability to step the executable to the position of the original entry point, and dump the entire executable. It also significantly affects program analysis and creates problems for rapid analysis. Therefore the only options available are to reverse engineer the decoding mechanism and manually decode the executable, or to use a dynamic method to extract the relevant information. The rest of this paper will focus on dynamic methods of program execution monitoring.

## 3. DEBUGGING REQUIREMENTS

Software armoring techniques all have a single common failure point, which is the processor must execute real machine code. Once this occurs an analyst should be able to observe and understand the code's functionality and capabilities. While there are methods such as RISE [16] that could add further difficulty the analyst's ability to observe and understand deobfuscated code execution in general most armoring techniques succumb to this fundamental situation.

Covert debugging is defined as the method of being able to single step and analyze a running program without indication to the debugee of this status. The requirements for a covert debugging system are as follows. First it should be able to handle a wide-variety of executables. This paper limits its scope to the context of portable executable PE files running under the Windows operating system. PE files were chosen as the majority of malware threats are present on the Windows platform. Given the different methods for protecting PE files, a system that could robustly handle each of these without difficulty was needed.

Efficiency is a prime concern. Given a sufficiently large collection of malware it is important to be able to extract relevant information rapidly without incurring too much overhead. The analyst's time is valuable therefore speed and automation are essential.

## 4. DYNAMIC INSTRUMENTATION

Dynamic instrumentation (DI) can be used to trace the exact execution of a debugged binary. This has many applications inside of computer architecture analysis as well as program analysis. DI tools such as *Valgrind* and *PIN* can provide insight into the execution characteristics of a program. For malware analysis Pin is especially useful.

Intel PIN is a highly configurable software tool for tracking the run-time execution of a program. Pin interfaces with the machine code via a series of callbacks that are registered on analysis startup. One can divert execution of a program at each instruction step. Memory access can also be monitored for reads and writes to addresses. All of these systems make it easy to monitor a running program, and divert execution if necessary.

## 4.1 Application to Packed Executables

Saffron uses PIN's dynamic runtime tracing to monitor the flow of executables. The results that are found are quite startling. First it is possible to find the original entry point for a packed executable. In section two we highlighted various methods used to protect and compress a running executable. By tracking the memory reads and writes, we can watch where the program modifies memory locations during the course of execution. Furthermore we can use this information to watch for executions inside of this written memory area. This is a highly likely candidate for an original entry point.

## 4.2 Results

The packers ASPack, UPX, FSG, and PeCompact, and TeLock were considered for the initial set of tests. These packers either compress or obfuscate a malicious binary. In each case except for TeLock, the use of a specialized PIN module was able to identify the original entry point for the packed executable. The results were then verified by manually reverse engineering the executable. The TeLock packed executable did not run properly when monitored using the PIN instrumentation library. Telock uses a memory CRC check prior to relinquishing control to the original entry point.

There are other problems with dynamic instrumentation. First it is extremely slow, especially when monitoring at the instruction level. Packers that use SEH for unpacking also present the same problems with PIN as they do with standard debuggers. Due to this it is necessary to move outside of the context of a userspace solution, and look at controlling execution from the kernel mode.

# 5. PAGEFAULT HANDLER DEBUGGING

The pagefault handler is an important part of modern operating systems. Its primary purpose is to assist the CPU's memory management unit in paging, or writing to disk, contents of the virtual memory space. It also performs tasks such as enforcing permissions for kernel mode versus user mode memory regions. Since it has a special place within the operating system, mainly controlling access to the memory of all the applications on a system, it can be used to control the execution of an executing program. The bulk of this work is a modification of the OllyBonE tool by Joe Stewart. This section will discuss the implementation of the new method, which has been named Saffron.

## 5.1 X86 Memory Management

The X86 architecture implements a combination of segmentation and paging mechanisms for implementing virtual memory. [15] Virtual memory is simply a method to create a protected address space for processes running on a computer system. It allows for a non-contiguous address space (physical memory) to be used as a contiguous one (virtual memory). The memory address that an application refers to must undergo a transformation to the physical address space of the main memory. The translation look-aside buffer (TLB) is primarily responsible for performing this lookup. The purpose of the TLB is to improve speed by using hardware to assist the lookup process for a section of memory.

## 5.2 The TLB Process

At its core, the TLB is an associative array for translated blocks of memory. When a virtual address is first referenced in a program, the TLB will look to see if it is present in its cache. If it is found the physical address is translated without further action. This is referred to as a "hit". If the address is not found it is referred to as a "miss". For the miss case, there are two common methods for memory management architectures. The first is to have the TLB walk the segment tables to check for the existence of a memory page. If the page is found in memory it is returned and entered into the TLB for future lookups. If the page is not found, or the page table entry (PTE) has any access flags set the page-fault handler is invoked. It is then up to the operating system to perform a software lookup of the memory address. If the OS can find no reference to the page of memory, then an exception occurs and the OS must handle it. [15]

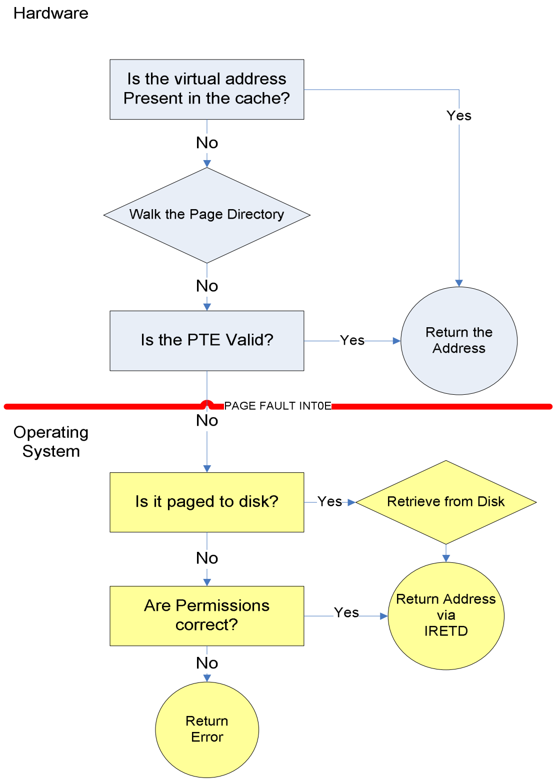

**Figure 1: Typical TLB / OS Interaction**

Each process has its own set of segments that are used. On a process context-switch, the TLB flushes its caching and begins the process of loading new memory translations. The actual characteristics of memory are controlled by a data structure called a page table entry (PTE). The PTE contains various flags for permission controls of the particular section of memory.

When the TLB begins the process of translating a virtual address to a physical one it first checks its local cache. If the cache does not contain an entry for the virtual address it then begins to manually traverse the page tables to find the virtual address in hardware. Once the page table entry has been found, it checks a series of fields to make sure that the permissions are correct. Once this is done the memory address is returned to the referencing instruction. Figure 1 illustrates this process.

## 5.3 Saffron's Pagefault Implementation

To implement the debugging of an executable, the program must allow for a stepping operation of the executable. The previous implementations are focused on stopping or diverting execution. Furthermore they are privileged enough to have knowledge of specific areas of memory to fault on. Shadow Walker uses its own memory that it can control in non-paged memory. PaX has low-level access to the OS primitives which control the memory management mechanism under Linux. OllyBonE relies on the reverse engineer tagging various areas of memory which it knows are valid. Unfortunately this pre-knowledge cannot and is not known at the time of execution for a generic application. Saffron consists of two distinct elements. First it uses a hybrid memory scanning and tagging mechanism to identify memory areas. Second a modified version of the page-fault trap handler is used to dispatch information on the page-faults.

The memory mapping algorithm is a brute-force on the entire process's virtual memory area. All memory addresses are enumerated based on the page boundaries. These addresses are then tagged with the supervisor bit so execution can be monitored. The page tag boundaries are useful in optimizing this marking process.

The page fault execution detection is very similar to the code used in Shadow Walker and OllyBonE. The modification to this method is to move from a faulting operation to a logging operation. The other modification is to allow the address to continue to execute once it has been logged instead of triggering the debug interrupt.

## 5.4 Results on Packed Executables

The same method was implemented as in section 4.1. Since there is no modification to the process' address space, there is no method for detection in the context of the executable. Monitoring of runtime execution allows for a coarse grain view of the executable. As expected this allows the detection of the original entry point on a packed executable.

There are some problems with this method. Since the page caches are aligned on a 4k boundary, there will be times when a memory read or write will go unnoticed by Saffron. Even with this method it is still possible to track an executable sufficiently enough to find out where the original entry point is located or to get a dump of the process's memory space with useful, deobfuscated information.

# 6. CONCLUSION AND FUTURE WORK

Dynamic instrumentation and pagefault handler subversion are extremely useful for tracking the execution of unknown or obfuscated executables. Since dynamic translation affords the most control over an executable, it should be the first method attempted when tracing runtime execution. If a particularly virulent and clever method is able to subvert that, the page fault execution mechanism is useful to track memory usages.

There is much work to be done in the methods of making the page fault tracking mechanism more robust. The Intel implementation of a split data TLB and instruction TLB causes problems with fully tracing the runtime execution using the page fault mechanism. One useful application of this method would be to apply it to a shifting frame decode unpacking method.

# 7. ACKNOWLEDGEMENTS

We would like to thank Lorie Liebrock for helping to hash ideas out about the page fault handler algorithm. Special thanks go to Bill Weiss for helping to debug Windows kernel code and for many useful brainstorming sessions. Thanks also to #vax and Jerry DeLapp for their help and support.

# 8. REFERENCES

[1] Rutkowska, J., *Beyond The CPU: Defeating Hardware Based RAM Acquisition Tools (Part I: AMD case)* – Blackhat Federal, February 2007

[2] Ferrie, P., *Attacks on Virtual Machine Emulators*, Symantec Advanced Threat Research

[3] Amini, P., Carrera, E., *Reverse Engineering on Windows: A Focus on Malware*, Blackhat USA 2006

[4] Miller, M., Johnson, K., *Bypassing Patchguard on Windows x65*, Uninformed Volume 3., January 2006

[5] Johnson, K., *Subverting Patchguard Version 2*, Uninformed, Volume 6, January 2007

[6] Stewart, J., *OllyBonE: Semi-Automatic Unpacking on IA-32*, Defcon 14, August 4, 2006

[7] PaX Team, *PaX*, http://pax.grsecurity.net/docs/pax.txt

[8] Sparks, S., Butler, J., *Shadow Walker: Raising the Bar for Windows Rootkit Detection*, Phrack 63, Volume 11, File 8

[9] Vasudevan, A., Yerraballi, R., *SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation*, Australian Computer Science Conference 2006

[10] Cifuentes, C., Waddington, T., Van Emmerik, M., *Computer Security Analysis through Decompilation and High-Level Debugging*, Workshop on Decompilation Techniques, pp.375-380, 8th IEEE WCRE (Working Conf. Rev. Eng.), Oct.2001

[11] Ma, J., Dunagan, J., Wang, H., Savage, S., Voelker, M., *Finding Diversity in Remote Code Injection Exploits*, Internet Measurement Conference, 2006

[12] Bayer, U., Kruegel, C., Kirda, E., *TTAnalyze: A Tool for Analyzing Malware*, EICAR 2006

[13] Robin, J., Irvine, C., *Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor*, Proceedings of the 9th USENIX Security Symposium, August 14-17, 2000.

[14] Quist, D., Smith, V., *Further Down the VM Spiral*, Defcon 14, Las Vegas, NV 2006.

[15] Intel Processor Programmer's Manual, Volume 3

[16] Barrantes, G., *Automated Methods for Creating Diversity in Computer Systems, Ph.D. Thesis*, University of New Mexico Computer Science Department, 2006