Scuola Superiore Sant'Anna
di Studi Universitari e Perfezionamento

Classe Accademica di Scienze Sperimentali
Settore di Ingegneria

Transparent and Efficient Instrumentation and Debugging of 32-bit Binaries

Tesi di Licenza di
Jacopo Corbetta e Alessandro Pignotti

Relatori:
Prof. Giovanni Vigna
Prof. Christopher Kruegel

Tutori:
Prof. Paolo Ancilotti
Prof. Giuseppe Lipari

Sessione 12 Luglio 2010

Abstract

Binary analysis is a notoriously difficult problem to tackle, especially in the context of malware research. Dynamic analysis systems execute the original binary in a secure, controlled environment. Instrumentation systems allow researchers to transparently modify the original code to perform sophisticated in-process tracking and analysis.

A vital requirement for analyzers is that their presence is transparent to the code being analyzed. With increasingly sophisticated detection techniques being employed by malware, this is a challenging task. We shall explore some of the problems arising due to the peculiar characteristics of the common IA-32 platform.

Another challenging task is achieving good performance without compromising transparency. While custom hardware solutions have been proposed in the past, we devised a solution that exploits the unique opportunity offered by the current state of transition between 32-bit and 64-bit architectures to allow efficient instrumentation on common PC hardware. A fixed mapping into the high address space allows elegant and efficient handling of indirect branches, a major source of overhead for traditional translation systems. Extended registers also relieve the need to continuously spill and restore those used by the program.

We built a prototype system that runs on off-the-shelf Windows PCs and is able to efficiently handle interactions with the OS, through a partial reverse engineering of the WOW64 subsystem.

It should be apparent that the binary analysis problem is related to the well-known platform virtualization problem, where single-process controlled execution is the user-space equivalent of efficient system virtualization. For this reason, we will attempt to use a unified terminology when referring to debuggers, tracers, instrumentation systems and even full-blown emulators.

Contents

Chapter 1

Introduction and motivations

The traditional medium for reasoning about computer programs is high-level source code. However, in many situations researchers and professionals have to deal with binary-only software artifacts. Binary analysis is the key component of a wide range of software engineering activities, from debugging dynamically-generated code to gathering performance metrics on production systems. An even more difficult problem is the one faced by malware researchers.

Ideally, an analysis system should be able to cope with hostile, unknown binaries, and do so without compromising the safety of its host system. The system should also provide tools that can gather all useful information, perhaps even allowing the researcher to craft his tools to deal with unforeseen problems. Finally, systems intended for online use should operate efficiently and avoid unnatural slowdowns.

To summarize these requirements, the researcher should be able to analyze the program natural behavior, keeping interferences from the analysis system at a minimum. A number of tools have been developed for analyzing binaries on the IA-32 architecture. In the rest of this chapter, we will give a brief presentation of the IA-32 architecture, highlighting the characteristics that make IA-32 binaries particularly resistant to analysis.

At their core, emulators, virtualization systems, application debuggers, binary instrumentation systems and even anti-virus software, all have to deal

with the same basic issues. We will review existing approaches and see if they meet the necessary requirements. It should be noted that efficient binary analysis is the user-space equivalent of the well-known *system virtualization* problem, often exposing the same problems and design challenges. To emphasize this fact (and to avoid confusion), we will always use the traditional *host* and *guest* terminology to indicate respectively the analysis system (i.e., debugger, tracer, emulator, ...) and the "*target*" binary being analyzed.

Chapter 2 shows how the present state of transition between 32-bit and 64-bit platforms offers a unique opportunity to explore the design of efficient instrumentation systems and the key design goals and choices of the one we built. Chapters 3-5 are each devoted to a specific problem and detail the solution we propose in our system.

At an advanced stage in the development of our system, we became aware of the existence of an Intel tool called StarDBT based on the same 32-bit to 64-bit translation technique. StarDBT can perform full-system emulation, which is beyond the scope of our work, but a user-space mode is also mentioned. While two papers have been published to date on the system [5, 27], it is hard to understand how much our system and StarDBT have in common because the actual system has not been made available to the public. We present our work in the belief that it provides a more detailed survey of the challenges involved in implementing fast user-space instrumentation, and possibly contributes original solutions to these problems.

1.1 Binary analysis

1.1.1 Static and dynamic analysis

While functionally equivalent to the source code, a production-ready compiled executable loses all the rich type information, all the annotations, and most internal organization of the source. Understanding the behavior of a binary can be a difficult and tedious task even when dealing with benign, non-obfuscated programs.

A possible approach to the analysis is to simply look at the target binary

code, without attempting execution. The researcher (possibly with the help of some heuristics) guesses the control flow and a *disassembler* decodes the binary dump into assembly instructions. Sophisticated disassemblers allow complex annotations and aid in analyzing the behavior of the target. Some can also try to identify common code patterns (such as exception handling code), or may even attempt to decompile the target into a high-level form that resembles source code. IDA Pro [14] is a well-known commercial disassembler with an extensive static analysis component.[1]

This kind of static analysis is obviously safe; the only risk is that binaries might be crafted to exploit known vulnerabilities in the disassembler. Since it is a completely offline procedure, performance is not an issue. It remains a widely-used procedure, and in the hands of a security expert it can be an effective means to understand the behavior of a (malicious) program.

However, malware authors are known to employ a variety of techniques to counter static analysis [18]. We shall see in the next section that the IA-32 architecture, the most common among computers connected to the present day Internet, presents many opportunities for obfuscation.

Dynamic techniques, instead, rely on executing the target in a carefully-controlled and instrumented environment. By doing this, dynamic analysis lets the code "reveal itself" through its behavior and its interactions with the environment. Dynamic analysis systems are in very wide use and take various forms, ranging from debuggers to user-level and system-level virtualization systems. They are also commonly used in the heuristic engines of modern anti-virus tools, which are deployed on millions of computers.

Our work will focus on dynamic analysis, both because of its flexibility and because it presents the biggest opportunities for automation and performance improvement.

[1] IDA is also able to operate as a traditional debugger and can therefore be used to conduct dynamic analysis. It also provides support for custom plugins.

1.1.2 Binary instrumentation

Many dynamic analysis tools exist. Some operate like debuggers, allowing for fine-grained control over the execution of a program, others trace the program behavior, make the manipulated data flow more evident, detect abnormal conditions, or assist in performance evaluation.

A particularly powerful approach is binary instrumentation. A binary instrumentation system allows the analyzer to modify, delete or integrate the original binary code in a completely transparent way. It should be noted that while it is theoretically possible to devise a static binary translator, such a system would need to be able to anticipate and handle all possible executable instructions, including those in dynamic libraries and in self-modifying code, an almost impossible task when faced with unknown binaries. Therefore, instrumentation systems aimed at binary analysis tend to be dynamic binary translators (DBT).

General-purpose frameworks like Pin [17], Dynamo [6], Valgrind [20], HDTrans [23] or even emulation systems like QEMU [4] can be adapted to perform a wide range of tasks. Whenever they encounter previously unseen code, their binary translation engine takes the original target opcodes, transforms them, adds the analysis code and then executes the modified instructions.

Some malware analyzers emulate the entire system and run the OS itself in a virtualized environment [1, 10], clearly separating host and guest environments, others sacrifice complete isolation to get substantial performance improvements [22]. Another possible choice is to operate in user-space, aiming at better performance and adaptability to a variety of environments.

The ability to execute code in the same context as the target binary allows very efficient inspection and manipulation of the guest environment. By contrast, relying on single-stepping or breakpoints implies a context switch for every event. Therefore, any fine-grained data collection is bound to significantly slow down the guest, perhaps to the point of making the analysis impractical. Moreover, unnatural program slowdown is in itself a possible detection avenue (see paragraph 2.1.1). Of course, the translation process itself causes overhead. We will analyze this issue in detail in Chapter 5.

Due to their flexibility and low intrusiveness, binary translators lend themselves to a lot of applications. Typical use cases for DBTs include instruction and data tracing, taint analysis, application profiling, instruction-set virtualization, and many others.

1.2 The IA-32 Architecture

A (perhaps undesirable) proof of the popularity of the IA-32 architecture is the amount of malware written for it. Therefore, we will focus on this very common architecture for most of our analysis. Most of our work is OS-agnostic and many considerations apply to other platforms as well, but to maximize real-world applicability our work focused on the Windows operating system (and, to a lesser extent, Linux).

The current 32-bit (x86) architecture is the result of a complex evolution. Many legacy instructions and facilities are retained, and often hamper efficient virtualization, especially at the system level.

The instruction set itself has many subtleties, which have often been employed to make static binary analysis impractical. The instruction length, for instance, is variable and no alignment restrictions exist. Therefore, code can be executed from any starting offset, with potentially very different results.

The original pagination system did not differentiate between code and data pages, making it possible to execute data on the stack or the heap. While this "feature" has seen some legitimate use (examples include gcc trampolines and the Microsoft ATL library [2]), it was also the basis of countless buffer-overflow exploits. Recent revisions of the architecture allow marking pages as No-Execute (NX). Luckily, PC operating systems have always required well-behaved applications to explicitly request the execution permission on memory pages (even though actual enforcing was impossible[2]), so this change is generally backwards compatible.

[2]Some patches for Linux and BSD managed to enforce this via complex segmentation-based tricks or by artificially setting the supervisor permission bit [26, 25].

1.2.1 System level virtualization

The IA-32 architectures offers the concept of Rings to protect sensitive and management code from applications. Only Ring-0 and Ring-3 are used by modern operating systems and usually the two modes are also called kernel mode and user mode. When running in kernel mode, it is possible to execute any instruction. When running in user mode, privileged instructions cause a trap. Those instructions include modification of the processor control registers, manipulation of permission flags, I/O involving restricted areas, and so on.

The basic requirement for efficient system virtualization, as formalized in 1974 by Popek and Goldberg, is that all control-sensitive and behavior-sensitive instructions are also privileged instructions. It would then be possible to execute an OS kernel in user mode. The host would conveniently receive control every time a sensitive instruction is executed. Due to a design flaw, the Intel architecture does not satisfy the Popek and Goldberg requisites. As an example that fits our discussion, the SGDT instruction returns the physical address of a control structure and works without traps even in user mode. Many virtualization systems (notably, VMware) set a different GDT[3] when running the guest, so a program can detect their use by comparing the SGDT-returned value with the well-known location on a physical system. Such detectors are often called "red pills" (in homage to *The Matrix*) and are becoming widespread among malware.

In recent years, both *Intel* and *AMD* introduced incompatible extensions (branded *VT-X* and *AMD-V*, respectively) to enable the execution of unmodified kernel-mode guests in user space. Both approaches are conceptually similar and we will call them *Hardware-assisted virtualization*. Basically these extensions fix the original flaws and make the architecture satisfy the Popek and Goldberg requisites. When running under hardware-assisted virtualization, every guest operating system is assigned a special data structure that virtualizes processor-state and gives the guest the illusion of running

[3]The Global Descriptor Table (GDT) is a data structure that contains access information about the various available memory segments.

alone on a physical CPU. Moreover, when running in this mode, all sensitive instructions cause a so-called *vm-exit trap*, which gives back control to the host. More recently the *Extended page table (EPT)* feature has been introduced. When this extensions is enabled, the usual pagination infrastructure translates virtual addresses to *guest physical addresses*. Those addresses are then translated to *host physical addresses* using another set of paging data structures. This enables guests to directly handle page faults and virtual memory, greatly reducing the amount of *vm-exit traps*.

1.2.2 User level virtualization

Besides the basic set of registers, the IA-32 architecture defines several other structures and flags that control user-space execution. As an example, setting the trap flag (TF) in the EFLAGS register enables single-stepping (that is, an exception is generated before executing each instruction). No permission is required to set the flag, and the instructions to read and set EFLAGS do not cause a trap. The situation is analogous to the SGDT one: if the host wishes to set the trap flag and use single-stepping for its own purposes, it must be careful to hide it from the guest program. Moreover, a known anti-debugging technique is to install an appropriate exception handler, set and reset TF a couple of times, and make sure that the appropriate exceptions are being generated (it is even possible for the debugger to mistake the trap for a user-requested single-step, confusing the user). Therefore, a truly transparent debugger should not only hide the use of the trap flag, but actually virtualize it and take care of accurately delivering exceptions to the target program.

Moreover, malicious programs can try to detect if their code has been instrumented or modified. Techniques usually boil down to some kind of code introspection to verify that the running code is exactly the one expected by the code author. The most common introspection is computing a checksum of the currently running code and comparing the result with a stored value. The x86 instruction set does not offer obvious ways of getting the current instruction pointer (EIP), however several tricks exist to obtain it. Typical techniques include reading it from the stack after issuing a CALL or triggering

an exception; in some cases, system calls will also leave the caller EIP in program-accessible registers. Systems that execute different code than the original must handle these cases and virtualize the instruction pointer.

More sophisticated detection techniques include exploiting OS-specific behavior and quirks. Modern applications, for instance, rarely use the segment registers, and they are usually re-used to concisely access thread-local storage and OS-maintained structures like the Windows PEB and TEB. Since their value is fixed and known, the OS doesn't need to actually save and restore them on context switches: they are always reset to the correct value. However, nothing prevents target code from setting them to some other value (as long as it specifies a valid segment selector, like the ones readily found in the other segment registers) and checking their value again after some instructions. If a context switch has taken place, the value will have been reset. Since context switches are much more common during a traditional debugging session, the target code can determine with some confidence if it is being subject to single-stepping. This is yet another argument in favor of in-process instrumentation, which avoids the context switch altogether, and can also replace the load and store from the segment register with appropriate virtualizing code. Once again, this is exactly the same technique employed by system-level virtualization systems.

Other techniques perform introspection through the OS, and range from straightforward calling of the IsDebuggerPresent Win32 API, to requiring the native NtYieldExecution call to return an error (indicating that no other threads are present), moving around code sections in an attempt to confuse the debugger, and so on [12, 9, 11].

Of course, dozens of such quirks exist, so a valuable strategy is to keep interference at a minimum, executing the guest in an environment that is as close as possible to the uninstrumented one, instead of attempting to replicate undocumented behavior.

1.2.3 The x64 architecture

The x64 architecture extends the x86 architecture in the following ways:

64 bit integers Arithmetic and logic operations work on full 64 bit registers.

Doubled register space The x86 architecture provides 8 general-purpose registers (namely: EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP; the latter is generally used as a stack pointer). When running in long mode, 8 more registers exist, named simply R8-R15. All these registers are or become 64 bit wide.

Doubled XMM register space Eight new 128-bit vector registers are introduced. The XMM registers were originally introduced with the SSE2 extensions, which is made a core part of the ISA in x64.

Larger virtual address space The address space is extended to 64-bit addressing. However, implementations can limit virtual addresses to 48 bits and require them to be in *canonical form* (having a certain number of high bits all set to 0 or 1), so some care should be taken in choosing addresses.

NX bit An additional permission bit to enable code execution on a page-by-page basis is available.

RIP-relative addressing Memory can be referenced by specifying an offset from the current instruction pointer (useful for position-independent code).

Besides some system features (VM86 mode, etc.), the only user-visible feature that was removed is segmentation. Segment limits are no more checked. However, FS and GS can still be used to specify a base address (this feature is typically used by Windows to reference its own process-specific and thread-specific structures).

The new 64-bit features are enabled only in long mode (*IA-32e*, in Intel reference material); real mode and protected mode continue to work as usual. In long mode, a compatibility submode is provided so that legacy binaries can run unmodified. The compatibility mode is virtually identical to the usual 32-bit protected mode, but the "extended" 64-bit state is preserved and the

switch is reasonably fast and triggered by simply setting the appropriate segment selector. All the usual x86 binary instrumentation tools can be used. They will not, however, be able to access the extended R8-15 registers or the address space beyond 4GB.

It should also be noted that the x64 architecture shares the same basic design as IA-32. Opcodes are generally the same and tend to behave in the same way modulo, of course, the extension to 64 bits of the registers. Moreover, the default operand size is still 32 bits. Immediate values are still limited to 32 bits (the exception being the immediate-to-register MOV) and are sign-extended if the operand size is 64 bits. 16-bit operations are still supported.

Memory addressing, on the other hand, defaults to 64-bit operation. A prefix allows forcing 32-bit operation.

In the next chapter, we will show how these features allow us to create an efficient instrumentation systems.

Chapter 2

Overview

We have seen how binary instrumentation can be a very powerful technique, but is a very complex and error-prone task on the IA-32 architecture. It can also cause significant performance overhead, especially if one wishes to run sophisticated analysis tools.

A major source of overhead comes from the fact that in traditional systems the analysis code and the original code have to compete for the scarce available execution resources (not only time, but also registers, stack and memory). Custom hardware solutions have been proposed [16, 7] that would alleviate this problem.

Such hardware might provide:

- extra registers invisible to the original program code;
- extra address space, not accessible by the original program code;
- an instruction set reasonably close to the original one, so that translation is not too cumbersome;
- execution speed comparable to the original systems;
- some help for indirect branches, like an alternate stack or extra hardware caching [16].

A look at the x64 architecture reveals that it readily satisfies the first four conditions. By exploiting the extended address space, we propose a

solution for indirect branches that does not require hardware modifications. By carefully exploiting the compatibility mode for 32-bit binaries offered by current operating systems, even OS interactions can be handled with low overhead.

2.1 Key goals

Stealthiness the ability to escape detection. Very good emulation and instrumentation tools exist, but they often rely on a special execution environment or make unusual modifications to the system, to the point that virtualization and sandboxing detection has become commonplace in modern malware. One of our goals is to create a tester that can run in a variety of environments and is small enough to be effectively hidden. While our system is not yet complete in this respect, we shall elaborate on this subject in the following chapters.

Speed As we mentioned, most existing tools have a high performance overhead. This not only makes using the system less practical, but can also become a route for detection. Moreover, even very mature instrumentation tools like Valgrind suffer from very pronounced overhead. As we will describe in detail in Chapter 3, our system can often make use of the extended registers instead of spilling and restoring the ones used by the program like many of the existing tools do.

Limited modifications to the system Binaries can run on a variety of environments, and the analyzer should not interfere too much. Besides making analysis more practical, keeping interferences at a minimum also reduces the risk of detection (see Section 1.2.2)

2.1.1 Bad performance as a risk for detection

It should be noted that adequate speed might be necessary to achieve stealthiness. A traditional anti-debugging technique, for instance, is to compare a

time-stamp counter before and after "interesting" code sections, with the aim of catching a (possibly human) analyzer single-stepping through the code.

While virtualizing the TSC can be enough to thwart simple detection attempts, it is possible to envision complex network-based techniques that make realistic wall-clock time performance necessary. A less sophisticated technique is to require very long and expensive computations, with the rationale that analysis systems having only a limited time available (anti-virus heuristic engines, for instance, or public web-based analyzers) will spend all their time in the "diversion" phase and will be unable to reach the actual payload. In the past, malware brute-forcing its own encryption has been reported [24, 3]. We estimate that timing attacks are very likely to be seen in the future.

Since time is a difficult factor to manipulate, it is hard to defend against sophisticated timing attacks. Should DBTs become widespread for malware analysis, red pills could start to detect the typical slowdown pattern caused by translation (a very slow first execution, followed by relatively fast subsequent executions). Abuse of indirect branches could force DBTs to take slow paths, and so on.

2.1.2 Limited modifications

A goal of the system we propose was to limit the modifications to the execution environment. On Windows, we employ a driver to free the high address space of the process, but after that the entire system runs in user-space and makes use of the unmodified WOW64 subsystem. Unfortunately, a kernel-mode component is needed (see Section 2.2.2) to access the entire address space. We have, however, kept its use to a bare minimum.

Few system calls (mostly memory-related) will need interception, the rest can interact directly with the OS.

The ability to install the analyzer on any PC is certainly convenient. It is also, as we hinted, an additional obfuscation factor. If the analyzer requires a perfectly clean environment or a recognizable system configuration, malware can simply start detecting that configuration, even if it could theoretically be

found "in the wild". As an example, many security researchers and analysis systems use VMware virtual machines. Therefore, malware started detecting typical VMware hardware and rapidly progressed to using red pills like the one we describe in Section 1.2.1. The ability to confuse analyzers far outweights the loss of some potential victims.

2.2 Overview of Prometheus

We will now describe the basic operation of Prometheus, the instrumentation system we built. While incomplete in many respects (noted in this document), we believe it provides proof of the viability of the 32-on-64 approach to binary instrumentation. Key design points will be expanded upon in the next chapters. In particular, we will describe here the basic approach to binary translation (fixed lenght), while in Chapter 5 we will propose a hybrid trace-based approach.

Prometheus runs on 64-bit versions of Windows (namely, Windows Vista) and uses the OS debugging support. As seen in Section 1.2.2, the attached debugger can be detected with some tricks. We chose user-space debugging for its simplicity, but we recognize that a kernel driver interfacing directly with the kernel-level debug facilities can offer substantially better stealthiness. We chose not to worry about Windows debugging specific problems (hiding the PEB debugging bits, for instance), since they are widely known and not immediately related to the instrumentation problem. We do care about issues related to binary instrumentation (e.g. EIP discovery), and in many cases our system achieves transparency.

While Windows was our primary target platform due to its relevance to malware analysis, it should be noted that the basic operation of Prometheus requires only pretty simple debugging facilities. Page-based execution permission enforcement is required. System call instrumentation code, of course, is completely OS-specific, as is most of Chapter 4. The WOW64 subsystem is taken here for granted, since we will make use only of the standard interface in this overview; a detailed description is postponed to Section 4.2.

To simplify testing, our implementation also offers typical debugger fea-

tures such as setting breakpoints, step-by-step execution of guest code, changing program context, etc. Prometheus implements the gdb server interface, so any regular 32-bit debugging client that can attach to a gdb server can be used with Prometheus. The implementation is pretty straightforward, given the other Prometheus features, so it will not be described.

We will now follow a typical execution of Prometheus.

2.2.1 Initial loading

We load the binary through the usual process creation calls, specifying the CREATE_SUSPENDED flag and registering as debugger for the guest process. The kernel will then send us notifications for interesting events from the very beginning.

Whenever a binary file is loaded, we detect its memory layout, with particular attention to code sections. For simplicity, we currently mark sections as either data or read-only code[1]. Ideally, we keep track of all executable pages in the process.

The initial phase in the life of a process is executing the loader code. As we will see in Chapter 4, most of the Win32 subsystem is implemented in user-space. To complete the creation of a regular Win32 process, Windows must map some system libraries in the address space of the process and hand control to them. This happens before any application code is run. In this phase, we allow the process to run without obstacles, since some important structures do not exist yet and system code can still be trusted. WOW64 DLLs will be loaded and the environment will be set up.

Immediately after process creation, we unset the execution permission on all guest code sections, so that the system will mark all those pages as NX.

[1]The PE format is very complex, and a sophisticated parser is needed to capture all information. Moreover, exotic combinations of permissions and section data are possible (TLS-related features even allow code to run before the specified program entry-point). We currently handle only the common cases. All Windows system libraries and most application software are covered by this, but malware is known to use corner-cases in the PE specification to confuse analyzers. Our current implementation is limited in this respect, but has a number of consistency checks in place that should at least detect possible implementation errors when handling benign code.

2.2.2 Preparing for instrumentation

When code execution hits a guest code page, an access violation exception is generated and intercepted. This ends the first phase. Program code is now ready to begin execution and so far the process has received no interference.

The first thing we need to do is allocate memory for our uses. A core objective of our analyzer is that memory accessible by the program is not modified. Therefore, all our structures and instrumented code sections need to reside in the high address space (beyond the 4GB mark).

The kernel-mode driver

There are standard APIs to perform memory allocations in the address space of another process, but Windows does not ordinarily allow high memory allocations in 32-bit processes. In the kernel, the loader recognizes the 32-bit nature of the applications and artificially allocates all high memory page. This is one of the very few cases in which the NT kernel actually discriminates between 32-bit and 64-bit processes, and is especially unfortunate since it required us to write a custom driver to remove the artificial allocation by manipulating the kernel structures. While this mangling can be seen as a limitation, currently the only clean solution would be to recompile the kernel and modify the loader. This is, anyway, the only purpose of the driver, and we specifically avoided adding other features to it, so that most of the code can still be adapted to run on other platforms.

The 32-bit helper

To limit interference with the system, our driver frees the high memory space only upon request. In order to activate the driver, a process must invoke CreateFile specifying a special file name. To free the high memory range for the guest, this request must be made *in the context of the guest process*. When we get the first access violation, we inject the code to do this in the address space of the guest process. We also change the guest execution context so that this helper code will be executed immediately. Since the

helper is injected in the low address space, we can do this using the regular debugger interface.

It should be noted that the traditional approach to code injection in Windows is to force the load of a DLL in the address space of the process and create a remote thread. This is, however, a very invasive technique. Not only all module, file mapping, and memory tables will be modified, but Windows will also notify all other DLLs loaded and execute the loader code, generating lots of debugging events and requiring lots of context switches.

Our helper just needs to call the CreateFile with the appropriate file name, and the driver will do the rest. While much less invasive, our simple binary injection approach requires that we do the linking ourselves. The NASM assembler allows us to easily create a "poor man import table": we determine the address of CreateFileA at runtime and write it in the table. Since the context is entirely determined by the debugger, we don't need to conform to calling conventions. We pass all parameters in registers and end the helper with a hard-coded breakpoint (`int3`).

Another 32-bit helper will be introduced in Chapter 4 to handle fast return from system calls.

Final allocation and preparations

Upon return of the 32-bit helper, we are free to prepare our structures in the guest memory, particuarly the instrumented code area (icode). This zone will hold a translation for each original program instruction and is organized as an "expanded" mirror of the low 4GB address space. Each byte in the original code corresponds to a fixed number of bytes (m) in the icode area. The icode corresponding to address a is simply $icode\_base + a \times m$. This very simple mapping is the key for our efficient handling of control flow, as we will detail in Chapter 5.

We then attempt to reserve all other high memory, to prevent accidental allocation to the process.

The final preliminary step is to inject and initialize our metadata structure. The structure contains the address of helper functions, the base address

of instrumented code and various spare memory locations that will be used to easily save and restore extended registers. Whenever our instrumented code is executing, register R12 will contain the base address of this structure, enabling compact access to its members.

Everything is now set up to begin code instrumentation. We will now mark all 32-bit code sections as non-executable (including those belonging to 32-bit system libraries) and begin the main execution phase.

2.2.3 Regular execution

Whenever code execution hits the original program code sections, we get an access violation exception due to the NX bit. Whenever this happens, we switch to the corresponding icode.

Switching from 32-bit code to 64-bit code

The Windows kernel treats all processes as 64-bit. The WOW64 subsystem is responsible for creating the illusion of a 32-bit system. We will expand on this in Section 4.2, but the basic idea is that the bulk of the program code runs in compatibility mode, while WOW64 switches the processor back to native mode when needed (e.g., to perform system calls) and then back to compatibility mode. It will also save and restore a copy of the 32-bit context.

We intercept code execution as soon as it attempts to execute 32-bit guest code. Switching to instrumented code therefore involves reading the guest context (for safety, we read both the real 64-bit context and the WOW64-maintained 32-bit context and verify they match) and carefully constructing the equivalent icode context. We just need to modify the instruction pointer ($RIP = icode\_base + m \times EIP$), the segment selector (to put the processor back into native mode) and set up the extended registers[2].

This switch must happen whenever 64-bit code calls back into 32-bit program code. Besides kernel callbacks due to, for instance, new thread creation,

[2]Remember that we want to use these registers for our own purposes. We must therefore save the state of extended registers R8-15 for future return to WOW64, since the WOW64 code uses the same trick.

this happens at every system call. Windows applications never call system services directly: instead, they call the system DLLs. The 32-bit DLLs, instead of performing actual system calls, perform far jumps to 64-bit *thunking* code (in our case, we do not even need to switch the processor mode[3]), which will adjust the parameters to system calls and generally maintain the illusion of a 32-bit system. Upon return from the kernel, WOW64 switches the CPU back to compatibility mode and attempts to resume execution of the original program code (WOW64 is unaware of our presence, and the only addresses it sees are 32-bit addresses in the original code sections); we will then receive the access violation exception and switch back the processor to executing instrumented code. Once again, Section 4.2 explains the mechanism in detail and shows how we can greatly speed up this process.

Translation

Code is translated on demand. All the instrumented code area is initially reserved but not allocated (committed). When a new icode page is referenced, we allocate it and fill it with hard-coded breakpoint instructions (`int3`, a single `CC` byte).

Stepping on the breakpoint in the icode area triggers the translation. The original instruction is parsed and a suitable translation is produced. If the original instruction is k byte long, we have a $m \times k$-byte long slot available to write the translation in.

As seen in Section 1.2.3, the translation is in most cases straightforward. In the next chapter we will cover this in detail.

Of course, as in any instrumentation system, additional tracking code can be added for each instruction. The remaining part of the slot is filled with NOPs (or a jump to the next slot, if the gap is very long).

Dealing with instruction aligment

x86 does not enforce any alignment for instruction opcodes and it is possibile, in principle to carefully craft binary code to have different semantics when

[3]We must, however, take care to restore the saved extended registers for WOW64

execution starts at different offsets (long immediate operands and addresses offer ample opportunity to hide code). This is a very powerful obfuscation technique and it is often used in malware. We do not currently support such misbehaving code. It could be handled by leaving a single `CC` byte (trap) in correspondence of every byte of the low address space (a `CC` every m bytes). Translated code should be crafted to jump over those control traps when executed from the "right" start. To find out if a received trap event requires regular translation or special handling for *in-the-middle* execution, two approaches are possible:

1. The address and lenght of each translated instruction should be kept in a hash table;

2. A special marking byte could be kept before or after the special `CC`s.

Using a marking byte makes the check very easy, but complicates instrumentation (that specific value must never be generated by the binary translator near the `CC` guard) and it consumes one byte of valuable icode space.

A table lookup, on the other hand, complicates run-time execution and can negatively affect performance. A goal of our approach was to specifically avoid doing table lookups at every branch, since it is a well-known source of overhead for traditional DBTs.

Chapter 3

Virtualizing the environment

The guest code can interact with many components. In many cases we need to treat these interactions carefully, to avoid detection and to maintain the original semantics. In this chapter, we will focus on interactions with memory and the processor resources (registers, etc.), while the interaction with the OS will be the subject of the next chapter.

3.1 Memory

One of the primary resources that the architecture provides to programs is memory. As every process on a modern systems has access to an apparently dedicated address space, a user space virtualization solution must provide an isolated, clean address space to the guest application. The isolation requirement is easily fulfilled using the memory protection mechanism provided by the hardware. It is not easy to provide a totally clean address space, as efficient virtualization systems will typically make use of use some memory in the address space of the target application, potentially exposing the system to detection and corruption.

Another issue that a virtualization system must face is the mixed nature of data and instructions. On the IA-32 architecture, self modifying code (SMC) is supported to a great extent and code pages may be modified using regular memory write instruction. The processor natively employs complex

circuitry to detect writes to instruction memory and invalidate instruction caches. Most of the existing virtualization solutions make use of some kind of dynamic binary translation, for example to guarantee isolation from the guest application. Since translated instructions are usually cached, complex mechanisms must be used to invalidate the cached translations when the original code is modified.

3.1.1 Transparency

As we said, keeping analyzer data structures in the same address space of the guest is a good way to reduce analysis overhead. No context switches or special system calls are required to update the metadata. Using shared memory pages, the host process can access metadata quickly and easily. Since analysis tools can require significant amounts of memory, it can be difficult to effectively "hide" the extra memory.

As an example, memory errors checkers such as Memcheck employ a technique called *shadow memory* to keep metadata for each byte of the program [19]. The metadata may be used to keep information about the value (for example the source of the data) or about the location itself (initialization status, number of accesses). The implementation of shadow memory is based on a coarsely grained memory pagination similar to the one provided by hardware. As the metadata page directory and tables have to be accessible from user space, they are allocated in the same address space of the process with loose access restrictions and so are fully exposed to corruption from the guest application. As an incomplete workaround, memcheck allocates data structure as far away as possible from the original data, to avoid unintentional corruption, which could be indeed common as the primary purpose of the system is to detect benign erroneus memory handling. Such a weakness, although not really an issue for Memcheck, would be not acceptable for a security tool. In a 2GB address space, it could be feasible to locate large foreign memory structures by simple scanning.

Instead, we prefer leaving the low 4GB address space untampered. We wish to guarantee that, were the program to read its entire 4GB address

space, nothing would change with respect to a regular execution without Prometheus[1].

Our approach exploits the extended 64bit address space offered by modern processors. By carefully guarding offsets, we can be sure that the guest (a 32-bit program) can never access more than the low 4GB address space. The metadata, translation cache and other host-related structures can be placed in the upper part of the address space, so isolation is guaranteed.

3.1.2 Example translations

The IA-32 supports many addressing modes, but fortunately only a handful of them require special handling in our system.

Let's review the IA-32 instruction format:

Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate

Of course, not all of these fields are always present. If the instruction involves memory, the ModR/M byte is present. Three bits of the ModR/M byte are part of the opcode or specify a second register operand (*reg/opcode* field). The two other fields (*mod* and *r/m*) specify the addressing mode. Several modes exist, involving different combinations of registers, an immediate displacement and/or a Scale/Index/Base (SIB) byte. Moreover, a segment override prefix can be specified; in modern operating systems with a flat memory model, this boils down to adding the segment base at the end of the computation.

When the processor runs in native 64-bit mode, the default behavior is to work on 32-bit data and 64-bit addresses. Simple indirect addressing (`[EAX]`) creates no problem: since any data manipulation defaults to 32 bits, the address in the register has already been truncated and zero-extended when it was loaded into the register. When using direct or offset addressing, the address could exceed the 4GB limit (for example when using a 32-bit negative

[1]We currently partially break this promise by injecting small 32-bit helpers. It should be noted that they can reside at an arbitrary address, so concealment is possible among system libraries code sections. After startup they are also not needed anymore except for fast system call handling, which can be disabled if maximum stealthiness is desired.

displacement). To avoid this, the translator needs to inject a special prefix that limits the address computation to 32 bits.

For example, suppose we encounter this addressing form:

```
FS:[EAX] + disp32
```

In this case, *mod* is 10 and *r/m* is 000 and the `FS` segment override prefix has been specified. The sum of a (truncated) EAX, a 32-bit displacement and the FS segment base can well exceed 2^{32}. In 32-bit mode, this address would simply wrap around; this addressing form is used to access data at a negative offset from a specified base. If we did not put any prefix, in 64-bit mode no wrap-around would occur and the instruction would erroneously operate on data over the 4GB limit.

Another, trivial, problem exists. In native 64-bit mode the combination of a *mod* of 00 and an *r/m* of 101 has been redefined: it now specifies RIP-relative addressing. It used to indicate that the whole address was a simple immediate 32-bit displacement. If we encounter this addressing choice, we translate it to a slightly longer form using a SIB byte (ModR/M specifies SIB+disp32, while the SIB byte does not specify any base or index).

Prometheus does not currently support 16-bit addressing, as its use is extremely rare and it is not available in native 64-bit mode. In principle, the translator could emit the instructions required to compute the correct address at runtime.

3.2 Registers and flags

Another important part of the execution environment are processor registers. In most cases, access to general-purpose registers does not need any special handling. However, registers are a particularly valuable and limited resource, and the analysis code must use them.

As we have already seen, a possible solution is to require a context switch whenever analysis code must run. The guest process will hand over control to the host process very frequently, however, and this can cause a significant performance hit.

If the analysis code runs in the same context, like in most instrumentation system, it must carefully save and restore the processor state every time it runs. Note that the processor state is not limited to the general-purpose registers. The condition flags must also be preserved, and since many instructions modify them, it can be tricky to write analysis code that does not require continuously saving and restoring them, even if the operation to be performed is very simple.

On the x86 platform, the LEA (Load Effective Address) instruction is particularly handy, since it allows complex operations without touching the flags. For example, suppose we need to subtract 4 from the `ESP` register (e.g. to simulate a PUSH).

Desired operation	**Using `SUB`**	**Using `LEA`**
`ESP -= 4`	`SUB ESP,4`	`LEA ESP,[ESP-4]`
Flags untouched	Flags affected	Flags untouched

Prometheus can use the extended register space to perform many operations without using the stack. Since they are preserved on context switches, extra registers are also handy for keeping pointers to structures, counters, etc.

3.3 Procedure calls and the stack

In native 64-bit mode, the stack must be 8-bit aligned. Since we do not wish to modify any memory below the 4GB mark, we have to emulate pushes and pops with regular memory access instructions by manually changing the stack pointer. It should be noted that the stack misalignment is a source of significant overhead.

Most security-oriented DBTs make sure that the stack is not altered. The OS sometimes uses the stack for its own purposes (e.g., exception handling), so for simplicity even speed-oriented DBTs can make the same choice [21]. However, it is in principle possible to completely reorganize program instructions and their stack and register use. Doing so could significantly boost our

performance, by keeping the stack aligned and exploiting the larger register space.

All branches must also be modified to point to the corresponding addresses in the instrumented code area. Static branches can be adjusted at translation time. Dynamic branches (including RETs) are modified so that the correct address is computed at runtime. We will return on this problem in Chapter 5.

Both issues must be dealt with when translating CALLs. In particular, the address pushed on the stack must be the one that the original code would have pushed.

3.3.1 Example translations

In our prototype implementation (see section 2.2.2), the `R12` register holds the base of our auxiliary structure (in the high address space). At offset 24 there is the base of the translated code. `R9` and `R10` are extended registers that can be used as temporaries with no harm for the guest. Note that the specific instructions used to do the mapping depend on the particular code multiplier in use (32, in the example).

Let's see some translations.

Original instruction:	`PUSH EAX`
Translation:	`LEA ESP,[ESP-4]` `MOV [ESP], EAX`
Notes:	`LEA` is used to avoid altering the flags

Original instruction:	`RET` *(near return to calling procedure)*

Translation:	`MOV R9, [ESP]` *(the return address)*
	`MOV R10, [R12+24]` *(is mapped...)*
	`LEA R9, [8*R9]`
	`LEA R9, [4*R9+R10]`
	`LEA ESP, [ESP+4]` *(...and removed from the stack)*
	`JMP R9` *(jump to the mapped return address)*

Notes:	We pop the 32-bit return address from the top of the stack. We jump to the corresponding slot in the instrumented code area.

Original instruction:	`CALL r/m32` *(call near, absolute indirect, address given in a 32-bit register or memory location)*

Translation:	`MOV R9, r/m32` *(the destination address)*
	`MOV R10, [R12+24]` *(is mapped)*
	`LEA R9, [8*R9]`
	`LEA R9, [4*R9+R10]`
	`LEA ESP, [ESP-4]` *(push the return address)*
	`MOV [ESP],return address`
	`JMP R9` *(jump to the mapped destination address)*

Notes:	The return address is always known at translation time: it is the address of the next instruction in the original code. Also note that we cannot push the return address immediately, since ESP could appear in the original *r/m32* specification.

3.4 Other processor resources

For a user-space application the processor state is composed of a subset of the globally available registers. For a 32-bit x86 application the accessible registers are the 8 general-purpose registers, the FPU registers (aliased as the MMX technology registers if MMX extensions are present) and 8 XMM

registers for vector operations (if SSE extensions are present) and a special purpose flags register.

When an application is running in 64-bit mode, it gains access to 8 new general-purpose registers and 8 new XMM registers. Those registers offer a very elegant and flexible solution for register shadowing (for tainting purpose) or to execute instrumentation payload without a significant performance loss caused by continuous register spilling and restoring. Unfortunately the x64 architecture does not offer extended register space for the FPU/MMX, so other solutions should be devised to instrument related code.

The `EFLAGS` register is definitely the most critical one and should be already partially virtualized to avoid host state percolating in the guest. As we have seen above, the guest should be only able to access the ALU conditions flags (such as Parity, Carry and Zero) and the direction flag for string operations. All the other flags are to be handled with care. As we saw in Section 1.2.2, one of the most critical points to avoid detection of the host from the target is the trap flag, which enables single step execution. Fortunately hiding those flags is straightforward in our translation architecture, as the EFLAGS register is only accessible as a whole using the `PUSHF` (and settable using `POPF`). The virtual state of the trap flag must be maintained separately in the tread-specific metadata. `PUSHF` must be translated so that the TF bit is read from the metadata, while `POPF` must operate only on the virtual flag.

3.5 The problem of self-modifying code

Self-modifying code (SMC) is especially hard to handle. The physical processor permits on-the-fly code modifications without explicit invalidation and dedicated circuitry exists to detect and correctly invalidated any cached structure the processor may be holding about the modified instruction memory. So any DBT caching translated code could be detectable by executing a code section, modifying the code, and executing it again: without proper detection of the SMC, the original, unmodified, code would be executed. In our architecture SMC handling could be done in three ways:

1. Adding a checksum verifier at the beginning of each slot to check if the original code has changed since the original translation. The overhead for this could be acceptable if instructions are opportunistically translated in traces (see Chapter 5);

2. Forcing all memory writes to invalidate corresponding icode memory areas. For example, whenever a byte is modified, the corresponding icode slot could be invalidated by writing a `CC` byte in it.

3. Completely disabling the icode cache, executing one instruction at a time without preserving the translation.

The basic problem here is that modern computers are designed to look like Von Neumann architecture machines[2]: data and code are treated in exactly the same way, and pages can be writable and executable at the same time.

Carefully applying page permissions can significantly alleviate this problem. There are two problematic situations:

1. **A previously-allocated data page gets execution permission, possibly multiple times:** we must invalidate all previously-translated icode sections involving bytes residing in this page.

2. **A page has write and execute permission at the same time**: each instruction can potentially modify many bytes in this page. Expensive SMC handling must be enabled.

Since Prometheus must virtualize page permissions anyway, we can write-protect all code pages. Whenever a write to a code page occurs, the debugger will get control and can manually invalidate code slots. Performance will suffer due to the additional context switches, but if few writable code pages exist performance will be significantly better that translating each memory write to a double write (the original write and the invalidation write), since the translation must be applied to all code. If memory writes in executable

[2]Internally, most modern CPUs look like Harvard architecture machines: code and data follow different paths and have separate caches. This is however not immediately visible, as self-modifying code is still supported, although it forces the CPU to disable many of its internal optimizations.

pages outnumber actual writable code execution, it could be better to switch to the third approach described above (no interception, but also no icode cache for writable memory pages).

Hopefully, common application programs will not need SMC detection. However, JIT compilers for interpreted languages like Java or .Net are becoming commonplace, so failure in handling SMC is not an acceptable option even for analyzers targeted at benign code.

3.6 Multithreading

Multithreaded code can pose some issues. First of all, a choice must be made regarding the translation engine: DBTs generally translate code on-demand, and multiple threads of execution imply multiple concurrent translation requests. Without locking, problems can arise. Our prototype uses the Windows debugger interface, which sends all program events to a single debugger thread, so we were almost forced to build a single-threaded engine.

A second problem is that some metadata must be kept on a per-thread basis. Since the code is shared among all threads, no fixed addresses can be hardcoded into the translated code. We can simply use one of the extended register (namely, `R12`) to hold the address of the auxiliary structure. The OS will take care to save and restore it as part of regular context switches, while we must set it whenever we manually hand control to instrumented code. The solution is similar to the one employed by Windows, which keeps the address of its own auxiliary structure (the *TEB*) in a segment base.

Of course, the analysis code itself must be concurrency-aware. For example, if an instruction trace is to be produced, the analysis tool could setup a separate journal for each thread, implement locking, etc.

Finally, the translated instruction themselves must maintain the original atomicity guarantees. For instance, if an increment instruction specifies the `LOCK` prefix, the instrumented code should offer the same guarantees. Luckily, the `LOCK` prefix can appear only on few instructions, and all of them translate to a single instruction in our prototype, so we can simply replicate the prefix.

If analysis code must be added, the situation becomes more complicated.

In principle, all the instructions referencing data manipulated by any original instruction must be executed atomically. It could be possible to integrate the CPU-provided `LOCK` prefix with a mutex maintained by the instrumented code. The whole translation of the instruction would be executed in a critical section. This would complicate instrumentation, but could be done with relatively few instructions and without requiring context switches in the uncontended case.

A simpler solution would be to artificially serialize all the guest threads, making sure that only one of them can be running at single given time. This is the approach employed in Valgrind [20]. This could reduce transparency (and performance), although it might be possible to convince the code that is running on a single-processor machine, where this behavior is normal.

Chapter 4

Mediating interactions with the OS

So far, we have analyzed the interaction of the guest with the basic execution environment provided by the architecture. However, far more complex interactions take place with the OS, its user-space libraries, and the kernel services it provides. We will first review the typical interface to OS-provided services. We shall also explore some of the ways to intercept and mediate these interactions. We will finally present the solution employed in our system, and show how we can exploit some characteristics of the WOW64 subsystem to efficiently handle Windows system calls.

4.1 Overview

Our review of how user-mode programs interact with the underlying operating system will reference Windows and Linux as they run on the 32-bit and 64-bit Intel platforms, but similar issues exist on most architecture.

4.1.1 Kernel calls

For obvious reasons of isolation, all operations involving I/O or resource allocation must be performed (or at least initially set up) by the OS kernel.

At the processor level, two obvious mechanisms exist to switch the execution to kernel-mode code: exceptions and deliberate calls.

Exceptions are generally expensive to generate and handle, but they can be transparent to user-mode code. Notable examples include paging: by setting restrictive permission on pages (or marking them as non-present), the OS will receive control when an access to the page is attempted. This can trigger actions that, for example, bring the requested page back into physical memory from the swap area and so on. The OS is also free to set liberal permissions so that no exception is generated for valid actions, and therefore they can occur with no speed penalty. User-mode execution can then resume at the point of interruption or at a pre-set resume trampoline. Kernels also generally provide the means to attach a debugger to a process, and will notify the debugger when exceptions occur. Therefore, while ascertaining the precise cause of the exception can be problematic, interception is generally not a problem.

In current operating systems, voluntary calls to kernel mode are generally performed through the SYSCALL or SYSENTER instructions, special lightweight versions of the common INT instruction. Kernels generally provide less than 255 of these system calls (as they are generally called in UNIX literature) or services (the name used in Windows documentation). Intercepting these calls is generally more problematic, especially since Windows does not notify the debugger when a kernel service is requested. Another issue is that these calls are the raw interface to the kernel code, which is usually poorly documented, can potentially change in every release, and does not always have a straightforward mapping to the documented API. Since a limited number of call indexes are available, many logical operations will be multiplexed in a single index, which can make interpretation even more difficult.

4.1.2 The user-space interface

Modern operating system loaders offer support for dynamic libraries (known in Windows as DLLs). Certain user-space libraries play the role of intermedi-

aries between the application program and the kernel services. Such libraries expose to the program the documented, public OS API (the Win32 API, for instance) and implement each function with the possible aid of one or more kernel calls. On UNIX systems, this task is generally performed by the C runtime library. On Windows a more complex system exists, with various DLLs implementing different parts of the API at different abstraction levels.

The Windows NT kernel supports application written for different APIs. For example, developers can use the POSIX API to create Windows applications that will run on any machine that has the POSIX subsystem installed. Ideally, a number of subsystems can exist, each providing its own user-space interface and implementing it using the native kernel services and a user-space helper process. In practice, the Win32 subsystem is the dominant subsystem; its DLLs and its helper process (CSRSS) are required for Windows to run. We will address only this subsystem.

For performance reasons, large parts of the API are implemented in user-space. Examples include the synchronization primitives on Linux, which are designed to require a kernel call only in case of contention, or the Win32 GetLastError function, which just reads a known location in the process virtual memory.

Moreover, in many cases the NT kernel will "call back" to user-space code (e.g., to create new threads, deliver asynchronous calls, et cetera). Such calls are made to boot-time determined fixed addresses. Therefore, ntdll (the library responsible for most user-mode to kernel-mode interactions) has to be loaded in the context of all processes, and always at the same address [15]. If address-space layout randomization (ASLR) is enabled, the address will change at every reboot, but will remain constant during system uptime. For different reasons, similar restrictions also apply to kernel32.dll and user32.dll.

4.2 The WOW64 subsystem

A particularly desirable feature of the x64 Windows kernel is that it is almost completely 32-bit agnostic. Only 64-bit calls exist, and the kernel makes no

difference between 64-bit and 32-bit processes[1]. Therefore, a compatibility layer must be provided to run 32-bit applications on 64-bit systems, namely the WOW64 subsystem[2] (*Windows-On-Windows64*). Official documentation is scarce, but the debugging support is excellent and the system resides entirely in user-space, so it is possible to get a good idea of how the system works.

WOW64 performs two main functions: switching the CPU to compatibility mode and *thunking*. Since the kernel is purely 64-bit, whenever control changes from kernel-mode to user-mode the CPU is in native 64-bit mode. A library, wow64cpu.dll, intercepts all control paths (including asynchronous procedure calls) that can cross the 64-32 boundary and does the appropriate switch.

WOW64 also performs thunking: many system services now expect 64-bit arguments, must be modified to keep the illusion of a 32-bit world, or have changed in other ways. The 32-bit version of ntdll has been modified so that, instead of performing a SYSCALL or a SYSENTER, a call to Wow64cpu.dll is made[3]. No other modifications are required except for other minor differences such as filesystem redirection. All other DLLs provided in the SysWOW64 directory could be (and generally are) unmodified Windows 32-bit DLLs.

Debugging is supported both in 32-bit to 32-bit mode (once again, emulated) and in 64-bit-debugger to 32-bit-client mode.

For comparison, Linux developers chose a much simpler approach: the kernel implements both the 32-bit and the 64-bit interface and, upon receiving control, can discriminate between the two. It can therefore support the regular 32-bit user environment, which can coexist with the 64-bit one (of course, separate copies of all libraries will be needed, including libc).

[1]For an important exception, see Section 2.2.2

[2]Technically speaking, WOW64 is not a distinct subsystem from the usual Win32 subsystem

[3]Two other DLLs exist, wow64.dll and wow64win.dll, perhaps originally to provide interfaces to the two parts of the Windows kernel.

4.2.1 Fast system calls

Summarizing what has been said, the flow for a typical system call is as follows:

1. The guest application calls a function in the relevant system 32-bit DLL (kernel32.dll, advapi32.dll, ...);
2. The system DLL calls the (32-bit) ntdll;
3. The 32-bit ntdll calls one of the WOW64 DLLs, through a slot in the TEB[4];
4. The WOW64 DLL performs the FAR JMP that would switch the processor to native 64-bit mode (in our case, we were already in 64-bit mode, of course; the FAR JMP is translated as a regular near jump);
5. It can now read the return address in the program from the stack, perform thunking, etc. It will eventually invoke SYSENTER or SYSCALL;
6. The kernel returns control to the WOW64 DLL that performed the call;
7. The WOW64 dll does a FAR JMP to the saved return address, switching the CPU back into compatibility mode and causing an access violation, since the address read from the stack will point inside one of the original program code sections (which are marked NX);
8. Prometheus will intercept the NX exception and perform the switch back to native 64-bit mode execution.

Note that point 7 and 8 involve two context switches. Excluding direct tampering with the system, this is probably the best optimization opportunity (the far branch for the return path is hardcoded in the 64-bit portion

[4]The TEB (*Thread Environment Block*, or TIB, *Thread Information Block*) is an auxiliary structure that Windows keeps for every thread. In 32-bit code, it can be accessed through the FS segment base. Besides holding Thread-Local Storage and system data, it is sometimes used to concisely call auxiliary functions without going through import tables.

of wow64cpu.dll: it is impossible to avoid this switch). We can replace the return stack pushed on the stack, forcing WOW64 to give us back control via the FAR JMP. More specifically, we can inject a 32-bit helper and have it change the back to native 64-bit mode without a context switch. Since WOW64 expects a 32-bit address on the stack, we are forced to inject the helper in the low address space.

4.3 Intercepting system calls

Capturing and logging system calls is a typical function of analyzers. As we have seen, "system call" is a bit of a misnomer, since it can apply equally to SYSENTER-like voluntary raw kernel calls and to regular "API-calls" to system libraries. Exceptions are not usually considered system calls.

Intercepting system calls at the system library level is tempting, because it often captures completely and concisely the intent of the application. On the other hand, since library code is in user-space, this code can be modified to confuse the analyzer. Modifying the library code is also a way to perform the logging itself. Inserting a jump to logging code as the first instruction of the API function (*detouring*) is the most common solution[5].

Even if interception is done at the API level, the logger should also be ready to intercept raw kernel calls, since malicious code sometimes attempts to escape detection by doing the SYSCALL/SYSENTER itself or by calling the so-called *Native API* exposed by ntdll.

4.3.1 Transparency

As we said, in general we prefer not to change system call results. In some cases, however, we might need to. Interception may be needed to thwart debugger detection techniques as shown in Section 1.2.2, although moving part of the debugger in kernel-mode might alleviate many of these problems.

[5]So common, indeed, that Windows system DLLs now prefix all their API function with a NOP prolog, presumably to workaround poorly-written detouring libraries that failed to correctly execute the instruction they replaced. It also makes it easier to write new detouring libraries.

Virtual memory manipulation is the other potential source of problem. As we have seen, we rely on page permissions to safely intercept all kinds of callbacks to the original code. The guest program is however free to call the virtual memory APIs to query and set page permissions. All those functions will need interception and page permissions must be virtualized. This could also make self-modifying code detection easier.

We haven't implemented this interception yet. It should not be very hard to do so, although the point for interception can probably sound unusual: since we know that system service calls can only happen in native 64-bit mode, the 32-to-64-bit switch is probably the best opportunity to examine the call parameters and, if necessary, invoke a helper to tamper with results. The host will likely have to maintain a "virtual" page table to keep track of what operations must be silently emulated and what operations must generate an exception.

Chapter 5

Efficiently handling control flow

With DBT technology becoming widely used, authors have started looking for ways to improve performance. Typical slowdown rates for modern systems range from 10% to 40% [27]. Hybrid 32-to-64-bit systems like ours and StarDBT have been designed with the explicit goal to explore the performance of this solution. A more traditional DBT with a strong focus on performance is FastBT [21].

5.1 Translation traces

Various factors can negatively impact DBT performance. Sheer code size is an obvious one: since all previously unseen code has to be translated, control must be switched to the DBT engine. In designs with a separate host process this implies a context switch. The translation itself can also be quite costly.

To improve translation performance, dynamic binary translators do not usually work on single instructions. Engines generally allocate some space in memory, translate several consecutive instructions, one after another, until a stopping condition is verified. The resulting translated code block is called a *trace* and can be executed right away. It is also generally put in a cache for later reuse. The last instruction of the trace must hand control back to the DBT engine or directly to another trace (in this case, traces are said to be linked).

Choosing the length of each translation unit involves balancing translation overhead and runtime performance.

Our basic approach, described in Chapter 2 (a fixed length slots for each byte), can be considered an extreme example: each trace is the translation of a single original instruction.

A more common approach is to stop whenever a branch is encountered. With this approach, we can be sure that only actually executed code is translated. However, the resulting translated code will contain lots of jumps to potentially far addresses, which decreases locality and reduces the effectiveness of caches.

The opposite strategy is creating traces that are as long as possible, in the hope that most branches will reference code within the trace. Compilers often generate short relative branches to implement loops and conditional statements. If the target address of those branches is within the trace, the instrumented code can simply contain the same branch with a slightly larger offset. This should keep performance of inner loops acceptable. If inner loops are spread among too many traces, performance will be abysmal. "Far" unconditional jumps are generally taken as the hint to stop the trace.

Creating long traces also has drawbacks. First of all, if jumping in the middle of a trace is not allowed, the DBT might have to translate the same code multiple times. Moreover, it is tempting to optimistically continue translating even after unconditional jumps. Unconditional jumps are often found in loops, and another instruction within the trace might reference code just after the unconditional jump. One must, of course, be ready to handle garbage possibly found after the unconditional jump (compilers align blocks of code, sometimes using unusual instructions like CCs for padding).

5.1.1 Our approach

Testing revealed that our original approach described in Chapter 2 did not yield good performance, since it reduced locality while significantly increasing the memory footprint.

We therefore devised a hybrid approach: we generate traces much like

traditional DBTs, but place them at fixed addresses. Traditional DBTs do not have the luxury of operating in an enormous address space, and therefore are forced to allocate traces dynamically. They keep tables that map addresses in the original program code to the address of the corresponding trace.

When Prometheus has to translate the instruction at address a in the original program code, it:

1. decodes the entire original instruction, determining its length (l);
2. determines its slot, which starts at $icode\_base + m \times a$ and is $m \times l$ bytes long;
3. starts translating instructions until said slot is full.

Note that in this case the code multiplier (m) can be set to a very high value so that long traces can be generated.

During step 3, we maintain a table with the starting address of each instruction we decided to translate in this trace, so that we can correctly adjust intra-trace branches.

We considered creating traces spanning multiple instruction slots but decided against it to avoid creating trace tables; the problem is similar to the one described in Section 2.2.3 for different instruction alignment, and similar solutions apply.

5.2 Branches

Branches[1] require very careful handling in DBT systems. As we have seen, sometimes we can simply adjust the branch offset so that it points to the

[1]We use the term in its most inclusive meaning. We consider a *branch* any instruction that can potentially defy the usual $EIP = EIP + lenght\_of\_current\_instruction$ rule. *Direct* branches point to statically known destinations, while *indirect* branches compute the destination at run-time. *Absolute* branches specify a fixed destination address ($EIP = destination\_address$), while *relative* branches specify an offset from the current instruction pointer ($EIP = EIP + destination\_offset$). *Conditional* branches take effect only if a certain condition holds, while *unconditional* branches are always taken. We sometimes use the word *jump* as a synonym of branch.

equivalent destination inside the trace. Otherwise, the branch must be modified to point to another trace.

If the destination address is known at translation-time, the correct trace can be linked right away. If the destination address points to code that has never been translated before, some systems temporarily insert the address of a *stub* that will call the appropriate translation code. After translating the new code, the stub address will be replaced with the address of the just-translated trace, so that further executions do not need to go through the stub. With this approach it is possible to translate code incrementally as new code paths become active, without pre-allocating traces [13]. Prometheus always knows the address at which a trace will be placed ($icode\_base + m \times address$), so no stubs are necessary.

Indirect jumps may not be handled this way. The usual solution is to keep a lookup table, and at run-time check if any trace already holds the target code. If the target code has not been translated yet, the translation mechanism must be triggered.

A possible optimization for common cases is to link the indirect jump to the requested destination trace when it is first translated and place an assertion to verify that subsequent execution has the same address. Such a system should fall back to the lookup case only if the assertion fails. This simple solution should work well with the indirect branches that are typically generated to handle C++ virtual inheritance.

More sophisticated solutions maintain a *shadow stack*, in an attempt to predict the destination of the branch, and have the destination block available without consulting tables. This works particularly well for regular call/return flows [21]. More complicated solutions are based on branch prediction systems, similar to the ones found in modern processors.

5.3 Performance

Almost all DBT systems keep a cache of the translated code to avoid re-translating code.

It is natural, therefore, to optimize DBT systems for *hotspot* translation:

frequently executed code sections (inner loops, key system code, etc.) that account for most of the execution time. Some systems, like HDTrans, go as far as incorporating profiling instructions into the generated code, to detect (and possibly re-translate) the hotspots.

However, the StarDBT authors have found that Windows GUI software in interactive workloads has less defined hotspots and translation overhead is significant.

It might be interesting to note that modern processors perform heavy branch prediction, and that mispredictions nullify the effect of the CPU pipeline. If no historical data is present, Intel processors will consider the direction of the branch, and will predict backward branches as taken and forward branches as not taken. It might be interesting for the DBT to allocate its traces so that this important hint information is preserved. A good side effect of our approach is that Prometheus always keeps branch hints intact.

We performed testing with the SPEC CPU2006 benchmark. Our prototype system is not yet mature enough to run the entire test suite, but from partial tests we can estimate that Prometheus currently imposes a 35% overhead. We conducted tests using the integer test suite (CINT2006) and encountered difficulties in running the Perl benchmark, possibly due to self-modifying code and the OMNet++ and gobmk benchmarks, likely due to bugs in our prototype. The libquantum benchmark is known to be incompatible with the Microsoft C++ compiler. The other nine benchmarks run fine, both using the test and reference workloads.

5.4 On-the-fly optimization

It is tempting to perform on-the-fly optimization of translated code. This can range from relatively simple transformations (e.g., StarDBT rewrites the original code so that it can use the extended x64 registers, thus reducing stack operations) to function inlining to eliminate indirect branches. Some systems even have performance improvement as their primary goal [8].

Since we envision malware analysis as a possible application for our system, we have chosen to preserve the original code behavior.

Chapter 6

Conclusions

We have shown that exploiting the current transition from 32-bit to 64-bit computing enables efficient and hardware-aided execution and instrumentation of untrusted 32-bit code in a controlled environment. Transparency and robustness are guaranteed by the back-compatibility of the 64-bit execution mode; in the few places where compatibility was broken, special translations emulate legacy behavior without incurring in much performance overhead. Interaction with the operating system is largely unfiltered, as this guarantees that the behavior of the target does not radically change even when it relies on undocumented behavior.

Even if 64-bit systems have been available for several years now, the adoption of 64-bit applications is far from complete in the Windows environment, and so it is for the malware scene as well; therefore, we foresee that our system will still be applicable in the medium term.

6.1 Possible future developments

In our work, we have attempted to recognize the fundamental parallel between system-level virtualization and transparent user-mode analysis. As system virtualization solutions have gained almost-perfect transparency by moving at a lower layer (using dedicated hardware support), we think that moving part of the analyzer in kernel-space could help avoiding many of

the problems we described and would make it substantially easier to achieve stealthiness.

Since in our approach we can freely use a whole new set of registers, it could be possible to write the analysis code in a high-level language and then compile it to native instructions than only reference the extended registers. The compiler would need to be modified to avoid altering the rest of the processor state. The generated payloads could then be simply inlined during translation, minimizing overhead while maximizing instrumentation development simplicity. Other modern DBT systems such as Pin employ a complex code rewriting system to efficiently mix original and payload code, so our system could be much simpler at the price of requiring a modified compiler.

We have outlined many of the basic issues in designing analyzers that are both robust and fast. Besides attempting to defeat timing-based attacks, we hope that such systems will enable heavy analysis to take place at realistic speeds. Well-designed systems might even be able to exploit multi-processor machines to gather data by instrumenting the guest code and analyzing the data on-the-fly on another processor.

Bibliography

[1] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org.

[2] SetProcessDEPPolicy Function (Remarks section). http://msdn.microsoft.com/en-us/library/bb736299.aspx.

[3] Aubrey-Jones, T. Behaviour Based Malware Detection.

[4] Bayer, U., Kruegel, C., and Kirda, E. TTAnalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)* (2006), Citeseer.

[5] Borin, E., Wang, C., Wu, Y., and Araujo, G. Software-based transparent and comprehensive control-flow error detection.

[6] Bruening, D. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Citeseer, 2004.

[7] Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., and Iyer, R. Defeating memory corruption attacks via pointer taintedness detection. In *IEEE International Conference on Dependable Systems and Networks (DSN)* (2005), Citeseer.

[8] Chen, W., Lerner, S., Chaiken, R., and Gillies, D. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)* (2000), Citeseer, pp. 81–90.

[9] corkami. Debugging Counter-measures. http://corkami.blogspot.com/p/anti.html.

[10] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[11] Falliere, N. Windows Anti-Debug Reference. `http://www.symantec.com/connect/de/articles/windows-anti-debug-reference`, 2007.

[12] Ferrie, P. Anti-unpacker tricks. In *Proc. of the 2nd International CARO Workshop* (2008).

[13] Gal, A., and Franz, M. Incremental dynamic code generation with trace trees. Tech. rep., Citeseer, 2006.

[14] Hex-Rays. IDA Pro. `http://www.hex-rays.com/idapro/`.

[15] Johnson, K. Why are certain DLLs required to be at the same base address system-wide? `http://www.nynaeve.net/?p=198`, 2007.

[16] Kim, H., and Smith, J. Hardware support for control transfers in code caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (2003), IEEE Computer Society Washington, DC, USA.

[17] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), ACM New York, NY, USA, pp. 190–200.

[18] Moser, A., Kruegel, C., and Kirda, E. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)* (2007), Citeseer.

[19] Nethercote, N., and Seward, J. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, p. 74.

[20] Nethercote, N., and Seward, J. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN Notices 42*, 6 (2007), 100.

[21] Payer, M., and Gross, T. Fast Binary Translation: Translation Efficiency and Runtime Efficiency.

[22] Sharif, M., Lee, W., Cui, W., and Lanzi, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM, pp. 477–487.

[23] Sridhar, S., Shapiro, J., Northup, E., and Bungale, P. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments* (2006), ACM, p. 185.

[24] Szor, P. Bad idea. *Virus Bulletin* (1998), 18–19.

[25] Team, P. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/`, 2003.

[26] van de Ven, A. New Security Enhancements in Red Hat Enterprise Linux v. 3, update 3, 2004.

[27] Wang, C., Hu, S., Kim, H., Nair, S., Breternitz, M., Ying, Z., and Wu, Y. Stardbt: An efficient multi-platform dynamic binary translation system. *Advances in Computer Systems Architecture* (2007), 4–15.