Protecting Against Unexpected System Calls

C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, J. H. Hartman
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{linnc,mohan,bakers,collberg,debray,jhh}@cs.arizona.edu

Abstract

This paper proposes a comprehensive set of techniques which limit the scope of remote code injection attacks. These techniques prevent any injected code from making system calls and thus restrict the capabilities of an attacker. In defending against the traditional ways of harming a system these techniques significantly raise the bar for compromising the host system forcing the attack code to take extraordinary steps that may be impractical in the context of a remote code injection attack. There are two main aspects to our approach. The first is to embed semantic information into executables identifying the locations of legitimate system call instructions; system calls from other locations are treated as intrusions. The modifications we propose are transparent to user level processes that do not wish to use them (so that, for example, it is still possible to run unmodified third-party software), and add more security at minimal cost for those binaries that have the special information present. The second is to back this up using a variety of techniques, including a novel approach to encoding system call traps into the OS kernel, in order to deter mimicry attacks. Experiments indicate that our approach is effective against a wide variety of code injection attacks.

1 Introduction

Code injection attacks, in which a remote attacker attempts to fool a software system into executing some carefully crafted "attack code" and thereby gain control of the system, have become commonplace. Such attacks can be broken down into three distinct phases. First, the attacker exploits some vulnerability in the software (a common example being buffer overflows) to introduce the attack code into the system. Next, the system is tricked into executing this injected code (e.g., by overwriting the return address on the stack with the address of the attack code). This then causes the various actions relating to the attack to be carried out.

In order to do any real damage, e.g., create a root shell, change permissions on a file, or access proscribed data, the attack code needs to execute one or more system calls. Because of this, and the well-defined system call interface between application code and the underlying operating system kernel, many researchers have focused on the system call interface as a convenient point for detecting and disrupting such attacks (see, for example, [5, 13, 17, 19, 29, 32, 35, 38]; Section 7 gives a more extensive discussion).

This paper describes an interrelated set of host-based defense mechanisms that prevents code injection attacks from executing system calls. The primary defense mechanism embeds information into the executable specifying the location and nature of each legitimate system call in the binary. This information plays two complementary roles. First, it allows the operating system kernel to verify the address from which a system call is made, thereby allowing it to detect system calls made directly from any injected attack code. Second, it supports a novel cloaking mechanism that allows us to hide the actual software trap instructions that trap into the kernel; this serves to thwart mimicry attacks by making it harder to discover library routines that trap into the kernel. This is backed up by a novel "code pocketing" mechanism, together with a combination of low level code obfuscation schemes, to thwart code scanning attacks aimed at discovering program code that will lead to system calls.

To be practical, the defense mechanism must work transparently with third-party software whose source code may not be available. Our binary rewriting tools analyze binaries and add system call location information to them, without requiring the source code. This information is contained in a new section of an ELF binary file. Our modified OS kernel checks system call addresses only if an executable contains this additional section. This makes our approach flexible: if an executable does not contain this section, the intrusion detection mechanism is not invoked. It is therefore possible to run unmodified third-party software as-is, while at the same time protecting desired executables; the use of binary rewriting means that an executable can be protected without requiring access to its source code.

The rest of the paper is organized as follows. Section 2 presents assorted background material. Sections 3 and 4 explain the proposed modifications to the Linux kernel and protected binaries. Section 4.2.1 describes how dynamically linked binaries can be handled using the same basic scheme. In Section 5 we summarize the results of deploying our implementation. Section 6 discusses limitations of our approach and directions for future work. Finally, Section 7 summarizes previous work related to intrusion detection, and Section 8 concludes.

2 Background

Our approach to intrusion detection and the steps we take to defend against mimicry attacks both depend on various aspects of the structure of executable files, the way in which system calls are made, and the mechanism for dynamically linking against shared libraries. For completeness, this section gives a high-level overview of the relevant aspects of these topics.

2.1 The System Call Mechanism

In most modern computer systems privileged operations can only be performed by the OS kernel. User level processes use the system call interface to request these operations of the kernel. This interface varies from system to system. The following describes the system call mechanism implemented in the Linux kernel, running on any of the IA-32 family of processors.

To invoke a system call, a user level process places the arguments to the system call in hardware registers `%ebx`, `%ecx`, `%edx`, `%edi`, and `%esi` (additional arguments, if any, are passed on the runtime stack); loads the system call number into register `%eax`; and then traps into the kernel using a special interrupt instruction, '`int 0x80`.' The kernel then uses the system call number to branch to the appropriate code to service that system call. One effect of executing the `int` instruction is to push the value of the program counter (i.e., the address of the next instruction) onto the stack; since this is done by the hardware immediately before control passes to the kernel, this value cannot be spoofed by attack code, and therefore serves as a reliable indicator of the location from which the system call was invoked.

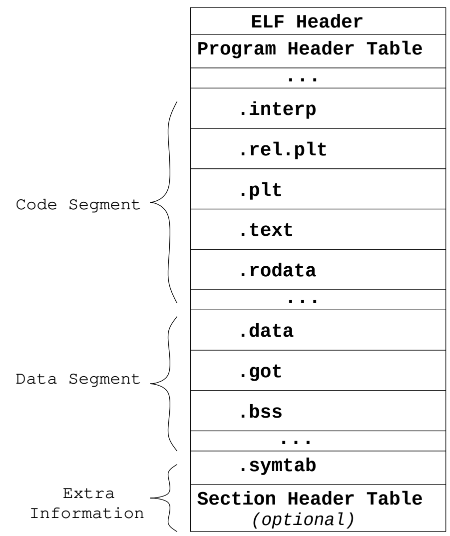


Figure 1: The structure of a typical ELF executable

2.2 Structure of an Executable File

The ELF (Executable and Linkable Format), first appearing in the System V Application Binary Interface, has become the most widely used binary format on Unix based systems. The structure of ELF binaries offers a great deal of flexibility and lends itself well to the embedding of auxiliary data [21].

Figure 1 shows the structure of a typical ELF binary. Conceptually, it consists of an ELF header, which contains basic information about the type and structure of the binary; some header tables that describe the various sections comprising the file; and a several sections, which contain the code, data, etc., of the program. Most executables have a `.text` section containing (most of) the executable code, a `.data` section that holds initialized data, an `.rodata` section that contains read-only data, a `.bss` section that contains uninitialized data, and a `.got` section that contains the global offset table (this is discussed more in Section 2.3).

The number and contents of the sections comprising an ELF file are not fixed *a priori*: we can add new sections containing auxiliary semantic information, provided that the relevant header tables are updated appropriately. We use this aspect of ELF files to embed, into each executable, information about the locations of system call instructions (i.e., the '`int 0x80`' instructions that trap into the OS kernel) in the file.

2.3 Dynamic Linking

When a binary is statically linked all functions referenced in the program are included, i.e., there is no need to load anything extra at runtime. A dynamically linked binary, however, may call functions that are not defined within the binary and are instead linked at runtime by the dynamic linker (ld.so). The details of this process are complex, and we discuss only those aspects of dynamic linking and shared objects that are central to this paper, namely, those which provide ways for attack code to execute a system call in a dynamically linked library.

Dynamically linked binaries are not loaded and run directly by the OS as are statically linked binaries. Instead, they contain an extra interpreter section (`.interp`) that contains the name of an interpreter binary (the dynamic linker) that should be run instead. The OS maps the dynamic linker into the executable's address space, then transfers control to it, passing it certain information about the target program, such as the entry point, location of the symbol tables, etc. The dynamic linker then scans the target binary's `.dynamic` section for any shared libraries on which the binary depends and maps their executable portions into the executable's address space. Private copies of any private data for the shared library is created by the linker and finally the linker passes control to the target program.

The default behavior of the dynamic linker is to resolve the address of each dynamically linked function when it is first invoked during execution (this is referred to as lazy binding). Two sections of the executable play a crucial role in this: the *procedure linkage table* (PLT) and the *global offset table* (GOT). Each dynamically linked routine has unique entries in the PLT and GOT. Initially these entries refer to the dynamic linker, so that when a dynamically linked function is first invoked control is transferred to the linker instead of the function. The dynamic linker uses the name of the function (accessible via the PLT linkage used to invoke it) to locate the function in the shared library's exported symbol table. The function's entry point address is then patched into the executable's GOT, the stack cleaned up, and control transferred to the target function. Patching the GOT entry causes subsequent invocations of the function to jump to the function instead of the dynamic linker.

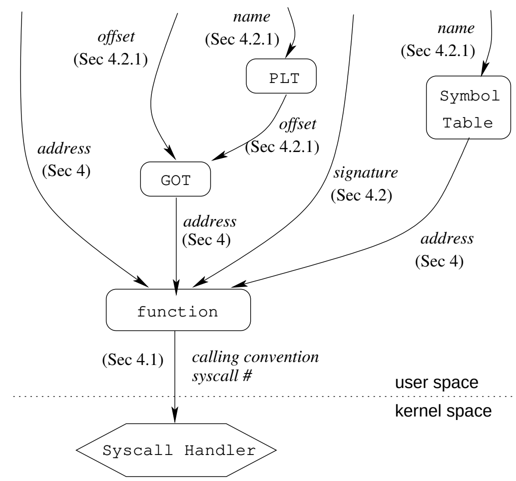


Figure 2: Attack model. Arrows represent information necessary to access data structures and functions represented by rectangles, with the ultimate goal of accessing the syscall handler at the bottom. Labels on arrows indicate sections where our preventative measures are described.

2.4 The Attack Model

This paper focuses on remote code injection attacks. We assume that the attacker has access to the source code for the application being attacked and to the methods we use to randomize the binary, but not to the particular instance of the randomized binary running on the host under attack. In other words, we assume some level of inscrutability, in that that the attacker has no way to determine the instruction sequence or layout of any of the system's programs, or shared libraries; we take advantage of this in our approach to detecting and preventing certain kinds of mimicry attacks. This assumption implies that any analysis of the executable being attacked must be done in an on-line manner by the attack code itself. In the extreme, the attack code could include a simulator on which to simulate and analyze the executable. We assume that the attack code is capable of such analysis, and do not assume *a priori* bounds on the amount of time or space that the attack code may use for such analysis. However, increasing the attack code's time and space requirements make intrusion detection easier; ideally, the bar is raised high enough that attacks have obvious symptoms. Finally, we assume that in order to do any damage to the system outside the compromised process, the attacker must make use of system calls.

Figure 2 is an illustration of the attack model. At the bottom is the system call handler inside the OS kernel. Executing this handler is the ultimate goal of the attacker. Rectangles represent functions and data structures; arrows represent information necessary to access them. For example, if the attacker knows the system call calling convention and proper system call number it can invoke the system call handler directly by synthesizing the proper code. Failing that, the attacker can invoke a function that performs the proper system call if it knows the function's address. Failing that, the attacker can get the functions's address from the symbol table or PLT/GOT if it knows the functions's name. And so on. Our methods of making the necessary information unavailable to the attacker are described in subsequent sections, as indicated by the labels on the arrows.

3 Adding Semantic Information to Executables

In essence, our goal is to distinguish system calls invoked illegally by attack code from those invoked legally as part of a program's normal execution. We begin with the simple observation that this objective can be achieved, in great part, by examining the address of the system call instruction: if this is not any of the known locations from which the application can possibly trap into the kernel, then it must be within attack code. This raises several issues, which we discuss in the remainder of this section and the next: (Section 3.1) how is the set of "allowed" system call instruction locations to be determined and associated with an executable? (Section 3.2) how should such information be used? (Section 3.3) how should dynamically linked libraries be handled? and, finally: (Section 4) what if the attack code co-opts an system call instruction that is part of the program code (or a dynamic library)?

3.1 Constructing Interrupt Address Tables

We use post-link-time binary rewriting to identify the address of each system call instruction in the executable (our implementation currently uses the PLTO binary rewriting system for Intel x86 ELF executables [31]).[1] This information is then added to the ELF executable as a new section, the *Interrupt Address Table* (IAT); the associated headers in the ELF file modified appropriately; and the file written back out. The IAT section is an optional component of an ELF executable, allowing executables that do not have this section to run as-is, albeit without the protections we describe. The information in the IAT consists of two values for each system call instruction found in the binary: (i) the address of the instruction immediately following the system call instruction; and (ii) the system call number associated with it.

Notice that there is enough information in the IAT entries that the system call numbers passed into the kernel by the system call now become redundant: the kernel could use the address pushed on the stack by the system call instruction to obtain the corresponding system call number from the IAT. This turns out to be very useful, as discussed in Section 4, for disguising system call instructions and thwarting mimicry attacks.

3.2 Using Interrupt Address Tables

We modified the Linux kernel to incorporate IAT information into the kernel data structure representing processes. When an executable is loaded, it is checked to see whether the executable file contains an IAT section. If it does, the contents of that section are copied into the kernel data structure for the corresponding process; otherwise, this field in the process structure is set to NULL. An executable that does not contain an IAT section is executed without any of the checks described in this paper; thus, third party software can be run as-is. The remainder of this discussion focuses exclusively on executables containing an IAT section.

When a system call instruction occurs during the execution of a process, the kernel checks that the address pushed on the stack by the '`int 0x80`' instruction appears in the IAT information for that process. A system call from an address found in the IAT is allowed to proceed; otherwise, a possible intrusion is signalled.

3.3 Handling Dynamically Linked Libraries

Unlike addresses in an executable binary, addresses in a dynamically linked library are not determined until the dynamic linker maps the library into the process's address space. This means that the dynamic linker must update a library's IAT after the library has been mapped, then make this updated IAT available to the kernel – the kernel cannot simply read the IAT from the library ELF file directly. The kernel then merges the information from the IAT into its internal data structure.

The dynamic linker uses a new system call to provide new IAT sections to the kernel. The arguments to this

[1] This assumes that there are no "hidden" system calls in the binary, i.e., system call instructions executed from dynamically generated code on the stack or heap, or in code that is dynamically decrypted and executed.

system call are the base address and size of the new IAT. If all libraries are mapped at program load time, the addresses of all system call instructions in the shared libraries will appear in the kernel level IAT for the process before it runs. The one exception to this is the interpreter (dynamic linker) itself, since it is a shared object and would not be able to make system calls before its own IAT section is loaded. This is not a problem because the kernel is responsible for mapping the interpreter into the executable (before the process begins execution), and it can therefore retrieve and patch the linker's IAT before the process begins to execute.

By default, the dynamic linker uses a lazy binding mechanism to map libraries – a library is not mapped until the process references it during execution. This makes the process vulnerable to mimicry attacks, and must be modified as discussed in Section 4.2.1.

4 Thwarting Mimicry Attacks

Mimicry attacks are attacks crafted to make the behavior of the attack code mimic the normal execution behavior of a program [37]. This allows such attacks to bypass intrusion detection systems that focus on detecting anomalous behaviors.

The IAT information makes it possible to identify any system call made from the injected code, since the addresses for such instructions will not appear in the IAT. To get around this the attack code must use a system call instruction that is already in the program: either part of the program code, or in a shared library. This section discusses the forms such attacks take and the steps we take to prevent them.

To use a system call instruction that is part of the program, the attack code must branch either (*i*) to the system call instruction itself, or (*ii*) to some location from which execution eventually reaches a system call instruction, e.g., some function in the standard C library. There are two possibilities. The first is that of a "known address attack," in which the attack code jumps to a fixed address that contains (or leads to) a system call instruction. The second possibility represents a class of attacks we term *scanning attacks*. Here, the attack code scans the application's code, starting from a valid code address (e.g., using the return address on the runtime stack), looking for a particular pattern of bytes; its aim is to identify a code address from which execution can reach a system call instruction. The pattern scanned for may be simply a byte sequence for a particular instruction, e.g., the 2-byte sequence `0xcd80` encoding the system call instruction '`int 0x80`,' or a longer sequence representing several instructions, e.g., some initial prefix of the `system()` library function. Such attacks can take a variety of forms, e.g.: set up the arguments to a particular system call, then scan for, and jump to, an `int 0x80` instruction; or set up the arguments to a particular library routine (say, `open()`), then scan for a byte signature for that routine and invoke it from the attack code. The first possibility listed above, that of known address attacks, can be foiled using a variety of techniques that make code addresses unpredictable, e.g., address obfuscation [3]. The remainder of this section therefore focuses on addressing scanning attacks. There are two distinct components to our approach: disguising system call instructions so that they are difficult to identify (Section 4.1); and making it harder to use pattern matching to identify specific functions (Section 4.2).

4.1 Disguising System Call Instructions

One weakness in existing executables is that system call instructions are easily identifiable, making them potentially vulnerable to scanning attacks, as described above. We can address this by making system call instructions harder to identify, by disguising them as other, less conspicuous, instructions (e.g., *load*, *store*, or *div* instructions). The idea is to use these other instructions to generate a trap into the kernel, e.g., by loading from an illegal memory address or dividing by zero, and letting the kernel decide whether the trap is actually a system call in disguise.

The IAT contains the addresses of legitimate system call instructions, making it easy for the kernel to decide whether or not a trap is a legitimate system call. The kernel checks the address of any instruction that causes a trap into the kernel against the IAT; a trap whose address is found in the IAT is processed as a system call, otherwise it is processed as a normal trap.

This scheme can be quite effective in disguising system call instructions. For example, since the Intel x86 architecture allows most arithmetic instructions to take a memory operand, an illegal address trap can be generated from a wide variety of innocuous-looking instructions, e.g., *add*, *sub*, *mov*, etc. Moreover, the particular instruction used to disguise a particular system call instruction in an application or library can be varied randomly across different systems.

From a practical perspective, disguising system call instructions in this manner makes it significantly harder for attack code to identify software trap instructions: instead of a handful of conspicuous '`int 0x80`' instructions, the attack code now has to contend with the possibility that pretty much any instruction in the program could potentially trap into the kernel. In even medium-sized programs, the number of such candidates could easily number in the hundreds of thousands. From a theoretical perspective, the problem of determining whether a given instruction—say, an *add* instruction with a memory operand—could cause a runtime exception is provably difficult: it is a straightforward reduction to flow-sensitive pointer aliasing, which is complete for deterministic exponential time [23].

4.2 Hindering Scanning Attacks

Once system call instructions become difficult to identify reliably, the attack code is forced to fall back on identifying specific functions that are known to lead to system calls. This section discusses ways to hinder this.

We can imagine two classes of such attacks. An attack might examine program metadata, e.g., symbol tables, to discover information about functions; Section 4.2.1 discusses ways to hinder such attacks. Alternatively, such an attack might scan the program text itself, looking for specific byte sequences. Given a function f in a program P, let $I_{f:P}$ be the shortest sequence of instructions (or shortest byte sequence) that uniquely identifies f within P. An attacker might examine his own copy of P, offline, to determine $I_{f:P}$, then craft a scanning attack that searches for this sequence. Sections 4.2.2, 4.2.3, and 4.2.4 discuss several different ways to thwart such attacks.

4.2.1 Symbol Information and Dynamic Libraries

One of the simplest ways to determine a function's entry point is to look up the function, by name, in the process's symbol table. The first and most basic step in defending against this, therefore, is to strip all symbol information from the binary. This is straightforward for statically linked executables, since (other than for debugging) symbol information is not needed after linking. It is not as straightforward for dynamically linked executables, however, because symbol information is fundamental to the default lazy binding scheme for resolving the addresses of dynamically linked routines (see Section 2.3). Removing symbol information from a dynamically linked executable would therefore break the standard lazy binding approach to resolving dynamically linked routines.

It does not seem straightforward to address this problem while using lazy binding for dynamically linked routines, since the standard lazy binding mechanism relies on the availability of symbol information. Our solution, therefore, is to abandon lazy binding and opt for "eager binding" instead. The idea is to have the dynamic linker resolve all GOT entries during the initial setup operations it performs, after the dynamic libraries have been mapped into the process's address space, but before transferring control to the main program. We can do this for the standard linker (ld.so) simply by setting the LD_BIND_NOW environment variable. Once all the GOT entries have been resolved in this manner, there is no further need for the symbols and relocations for the shared libraries, and they may be discarded. While this can potentially increase the startup time for a process, we believe that its impact will be small.

Conceptually very similar to the idea of scanning a dynamically linked executable's symbol table is that of scanning a loaded shared object's symbol table. To address this problem, we add a little extra functionality to our wrapper linker (see Section 3.3). After linking is finished, either just before or directly after we discard the symbols in the executable, we unmap the memory regions in the shared libraries that contain symbol and relocation information. This makes them inaccessible to attack code; any attempt to scan these regions of the library results in a segmentation fault that can be caught and flagged as a potential intrusion.

A final problem is that the GOT (directly) and the PLT (indirectly) identify the entry points of all library routines needed by a dynamically linked executable. This can allow an attacker to obtain a function's entry point by exploiting knowledge of the structure of a process's GOT. For example, if a program uses only a few dynamically linked library routines, the number of GOT entries will be correspondingly small. In such cases, an attack may be able to guess the correct entry point for a desired function, with high probability, simply by randomly choosing an entry in the GOT. A simple protective measure to address this is to introduce many fake entries into the GOT and PLT. Because the GOT and PLT usually account for only a very small fraction of the size of an executable, the space impact of such fake entries will usually be small. A second problem is that the GOT may, by default, have a predictable layout, i.e., the same function may lie in the same GOT slot in many or all copies of the executable. This would allow an attacker to execute any of the GOT resident library functions without any guesswork. This can be handled by randomizing the order of the entries in both the PLT and GOT.

An alternative approach to handling the problems introduced by dynamic libraries is to abandon dynamic linking altogether in favor of static linking. The SLINKY project [7] has shown that with very minor effort (a small tweak to the operating system kernel and some additional system software) the overhead traditionally associated with static linking can be largely eliminated. The resulting statically linked and stripped binaries will contain no symbolic information exploitable by the adversary.

4.2.2 Dead and Useless Code Insertion

A simple way to disrupt attacks that scan for specific byte sequences is to insert randomly chosen instruction sequences into the code that change its contents but not its semantics [14]. Examples of such instruction sequences include: `nops` and instruction sequences that are functionally equivalent to `nops`, e.g., '`add $0,` *r*', '`mov` *r*`,` *r*', '`push` *r*`; pop` *r*', etc., where *r* is any register; and arithmetic computations into a register *r* that is not live. In each case, we have to ensure that none of the condition codes affected by the inserted instructions is live at the point of insertion. It is worth noting that some advanced viruses, e.g., encrypted and polymorphic viruses, use a similar mechanism for disguising their decryption engines from detection by virus scanners [33, 40]. The approach can be enhanced using binary obfuscation techniques [22].

The higher the frequency with which such instructions are inserted, the greater the disruption to the original byte sequence of the program, as well as the greater the runtime overhead incurred. One possibility to determining a "good" insertion interval would be to compare the byte sequences of all the functions (and libraries) in a program to determine, for each function, the shortest byte sequence needed to uniquely identify that function in that program, and thereby compute the length of the shortest byte sequence that uniquely identifies any function. Any insertion interval smaller than this length would be effective in disrupting such signature-based scanning attacks.

4.2.3 Layout Randomization and Binary Obfuscation

Code layout randomization involves randomizing the order in which the functions in a program appear in the executable, as well as randomizing the order of basic blocks within each function [14]. In the latter case, it may be necessary to add additional control transfer instructions to preserve program semantics.

In principle, the attack code could overcome the effects of layout randomization by, in effect, disassembling the program and constructing its control flow graph, thereby essentially reverse engineering the program. While this is possible in principle if we assume no limits on the time and space utilization of the attack code, it would require the injected attack code to be dramatically larger, and more sophisticated, than attacks commonly encountered today. Moreover, such reverse engineering by the attack code can be thwarted using binary obfuscation techniques [22], which inject "junk bytes" into an executable to make disassembly algorithms produce incorrect results.

4.2.4 Pocketing

Another approach to thwarting scanning attacks is to divide the address space of the executable into noncontiguous segments, separated by "pockets" of invalid addresses. If the attack code accesses one of the invalid address pockets, it generates a trap into the kernel that can be recognized as an intrusion. On modern virtual memory systems, where memory protection is typically enforced at the level of pages, such pockets must appear at page boundaries and occupy an integral number of pages.

There are two distinct approaches creating such discontinuity. First, we can separate the code section into many segments, assigning to each successive segment a load address which leaves a gap from the previous segment's ending address. Second, we can create several gaps (via code insertion) in the executable sections and unmap them at runtime. The first approach has the disadvantage that the program header table will contain the exact addresses where pockets begin and end, which may introduce a vulnerability if the attacker happens to find the location of the program header table. The advantage of this scheme, however, is that the physical size of the executable on disk will experience only a minimal increase. The second approach has the disadvantage that the physical size on disk can increase dramatically. However, it offers the advantage that a careful implementation can actually hide the code that does the unmapping within the pockets themselves, preventing an attacker from discovering the location of the executable's pocket layout.

A straightforward approach to inserting pockets is to simply insert them at arbitrary page boundaries, adjusting in the obvious way any instruction that happens to span the page boundary, and inserting an unconditional jump to branch over the pocket. This approach is appealing because it introduces virtually no increase in memory requirements for the application. The unconditional branches, however, might act as an indicator of a valid continuation address that an attacker might follow to "jump over" pockets. An alternative approach, used in our implementation, is to insert pockets in locations where no modifications to control flow are necessary, namely, between functions. Since function boundaries are not guaranteed to lie on page boundaries, this approach requires adding some padding into the executable, which increases its memory footprint.

5 Experimental Results

We integrated our ideas into *plto* [31], a general purpose binary rewriting tool for the Intel IA-32 executables. Our tool implements all of the ideas described in this paper, with the single exception of the handling of dynamically linked libraries (Section 3.3); *plto* currently handles only statically linked binaries. Our tool takes as input a statically linked relocatable binary, and outputs the executable that results from performing intraprocedural layout randomization, `nop`-equivalent insertions, pockets insertions, or system call obfuscation, in various combinations determined by command-line arguments.

Our experiments were run on an otherwise unloaded 3.2 GHz Pentium 4 processor with 1 GB RAM running Fedora Core 1. All kernel modifications necessary for this intrusion detection system were implemented in the Linux kernel, version 2.6.1. The changes required to the kernel were minimal, spanning only a handful of source files, including the file containing the trap handler entry code, the file containing the ELF specific loader module, and the files containing the main task structure definition and task structure handling routines.

5.1 Design of Attack Experiments

One simple approach to evaluating the efficacy of our approach to detecting code injection attacks would be to subject it to several currently known viruses/worms. There are two major problems with such an approach. First, many different attacks may exploit the same kinds of underlying software vulnerabilities (e.g., a buffer overflow on the runtime stack), which means that the number of "known attacks detected" need not have any correlation with the variety of vulnerabilities that an IDS is effective against. Second, such an approach would completely ignore attacks that are possible in principle but which have not (yet) been identified in the wild. For these reasons, we opted against relying on known attacks to evaluate the efficacy of our approach. We decided, instead, to use a set of carefully constructed synthetic attacks, whose design we describe here.

We begin by observing that our work assumes that the attack code has been successfully injected into the system and then executed, and aims to prevent this executing attack code from executing a system call. The nature of the exploit by which the attack code was injected and executed—be it via a buffer overflow on the stack, an integer overflow, a format string vulnerability, or some other mechanism—is therefore unimportant: we can pick any convenient means to introduce "attack code" into a running application and execute this code. Furthermore, the particular application used for the attack is also unimportant, as long as it is a realistic application, i.e., one that is of reasonable size and which contains some appropriate set of system calls which we wish to protect. Accordingly, our efficacy experiments use a single vulnerable application, and a single code injection method, to introduce and execute attack code; this attack code varies from experiment to experiment and attempts to use a variety of different approaches to executing system calls. By using a carefully crafted collection of attacks in this manner, including both direct invocation of system calls using an '`int 0x80`' instruction in the attack code, and mimicry attacks involving scanning, we can gauge the efficacy of our approach to a wide variety of attacks.

We used the *m88ksim* program (from the SPEC-95 benchmark suite), a simulator for the Motorola 88000 processor, as our attack target. The program is roughly 17,000 lines of C code, which maps to a little over 123,000 machine instructions over some 835 functions (compiled with *gcc -O3* and statically linked). We chose this program because it makes use of several potentially dangerous library calls, including *open* and *system* (which eventually makes the system call *execve*). We simulated direct attacks, i.e., where the injected code contains the '`int 0x80`' system instruction for the system call it attempts to execute, by injecting the attack code onto the runtime stack and branching to it; the mimicry attacks, which involved various different ways to locate and branch to system calls in the library routines, were written in C and linked in as part of the program.

5.2 Efficacy

This section discusses the specific classes of attacks we tested, and the outcome in each case.

5.2.1 Injected System Call Instructions

The first class of attacks we consider execute a system call instruction directly from the injected attack code. In practice, such an attack might result from injecting executable code onto the stack or the heap and then branching to this code (e.g. the Morris worm). Our test that represents this sort of attack uses a simulated buffer overflow, where instructions are first pushed onto the stack, then executed by jumping into the code on the stack. The instruction sequence so injected contains a system call instruction '`int 0x80`' to invoke the system call, preceded by some instructions to set up the arguments to this system call.

Our tests show that such attacks are completely prevented via the interrupt verification mechanism proposed in Section 3. Upon executing an interrupt from any location not found in the IAT, the operating system correctly declares the interrupt malicious and takes appropriate action.

5.2.2 Known-Address Attacks

Since each binary is randomized on a per-install basis, as described earlier, we assume that the attacker is unaware of the absolute address of any particular function or instruction in the binary. The Code Red virus and Blaster worm are examples of known-address attacks that are thwarted if addresses are randomized. Bhatkar *at al.* have demonstrated the efficacy of such randomization techniques against this class of attacks [3]. We therefore did not separately examine known-address attacks in our experiments.

5.2.3 Scanning Attacks

We examined several scanning attacks that used pattern matching to try and discover the locations of valid system call entry points. Under the assumption that library code addresses have been randomized, e.g., via address obfuscation [3], such attacks must discover the locations of suitable system calls as follows:

1. *Identifying code that will eventually lead to the desired system call.* The attack code can scan for a known sequence of instructions from the code for that system call or a (library or application) function that invokes that system call.
2. *Identifying the appropriate system call instruction directly.* The attack code can scan for a system call instruction '`int 0x80`.' There are two variations on this approach:
   (*a*) identify a location where the appropriate system call arguments are set up, followed by the system call instruction; or
   (*b*) identify just the system call instruction, whereupon the attack code itself sets up the system call arguments appropriately, then branches to the system call.

We devised a synthetic attack representative of each such class of attacks:

1. As a representative attack that attempts to scan the code to find a known code signature, we used an attack that looks for a 17-byte sequence that comprises the first basic block (eight instructions) of the `execve` system call:

   ```
   55              // push %ebp
   b8 00 00 00 00    // mov  $0x0,%eax
   89 e5           // mov  %esp,%ebp
   85 c0           // test %eax,%eax
   57              // push %edi
   53              // push %ebx
   8b 7d 08        // mov  0x8(%ebp),%edi
   74 ff           // je   8076d26
   ```

   Note that there is nothing special about this particular byte sequence, other than that it happens to be one that is known to lead to an `execve` system call. We could have just as easily chosen another byte sequence corresponding to code for a suitable system call.
2. We used the following attacks to scan for system calls directly:
   (*a*) To identify code that sets up the system call arguments and makes the system call, we used an attack that scans for a 6-byte (two instruction) sequence to load the value `0xb8`, the system call number for the `execve` system call, into register `%eax`, followed by a system call instruction:

   ```
   b8 0b 00 00 00 // movl 0x$b8, %eax
   cd 80          // int $x80
   ```

   (*b*) To identify system call instructions, we simply looked for the two-byte (one instruction) sequence

System Call	*Time w/o IAT* (μsec)	*Time with IAT* (μsec)	*% Increase*
getpid	0.71	0.96	35.2
open	19.58	19.77	1.0
read	95.75	98.19	2.5

Table 1: Effect of IAT checking on an individual system call.

```
cd 80          // int $x80
```

Each of these synthetic attacks was unsuccessful against the implemented intrusion detection measures, namely use of the IAT in combination individually with each of disguising system call interrupts, `nop`-equivalent insertion, pocket insertion, and layout randomization. Attacks in category 1, which attempted to find code that would eventually lead to a system call, failed because of layout randomization and `nop`-equivalent insertion, which disrupted known byte sequences throughout the code. Attacks in category 2(a) and 2(b), which attempted to find the system calls directly, failed because the system call instruction was disguised, as described in Section 4.1.

5.3 Cost of the IAT Mechanism

There are two aspects to the cost of the underlying IAT mechanism: the incremental cost for an individual system call, and the impact on the overall performance of realistic applications. For these evaluations, our benchmark programs were compiled using *gcc* version 3.2.2, at optimization level `-O3`, with additional command-line flags to produce statically linked relocatable binaries. These binaries were then processed using our tool, described above. Execution times were measured using the *time* shell command. Each timing result was gathered by running the program 5 times, discarding the lowest and highest execution times so obtained, and averaging the remaining 3 run times.

To evaluate the effect of IAT checking on an individual system call, we measured the time taken to execute a lightweight system call (*getpid*) and two moderate-weight ones (*open, read*), with and without IAT. In each case, we used the `rdtsc` instruction to measure the system time taken to make each call n times in a loop (we used $n = 10,000,000$ for *getpid*, 100,000 for *open*, and *300,000* for *read*), and divided the resulting time by n to get the average time for a single call. We repeated this 10 times for each system call, removed the highest and lowest run times, and averaged the remaining eight run times. Table 1 shows the results.

Not surprisingly, *getpid* experiences the largest percentage increase from incorporating IAT checks in the kernel, but the actual increase is quite small, about 0.25 μsec per call on average. The additional runtime overhead for *open* and *read* are quite small: 1% for *open* and 2.5% for *read*. The reason *read* experiences a larger increase than *open* is that in the program we used, it happened to appear later in the IAT, which—because of the naive linear search currently used by our implementation—led to a larger search time.

To evaluate the effect of IAT checks on realistic benchmarks, we used ten benchmarks from the SPECint-2000 benchmark suite.[2] The results are shown in Figure 3. It can be seen from this that the effect of adding IATs on realistic applications is quite small: on average disk file sizes increase about 0.11%, total memory size by 0.45%, and execution time by 1.7%. This is not surprising, since for unmodified executables, the kernel executes only a few instructions per system call to discover that a process has no associated IAT and proceed without any further attention to verification. The overhead associated with executing binaries with the protection system enabled is only slightly higher. One reason for only such a small increase in runtime is that system calls compose such a small fraction of the overall runtime in general due to the low frequency of their occurrence.

5.4 Cost of Transformations to Thwart Mimicry attacks

5.4.1 Time Cost

The effect of using various techniques for thwarting mimicry attacks on execution time is shown in Figure 4(a). Pocketing incurs an overhead of 2.8% on average. The reason for this small overhead is that the unmapped pages inserted are loaded into memory only once and are never executed, There is, however, one program, *crafty*, for which pocketing incurs a significant overhead, of around 13.5%. This turns out to arise, not from the

[2] We were unable to build two other benchmarks in the suite, *perlbmk* and *eon*.

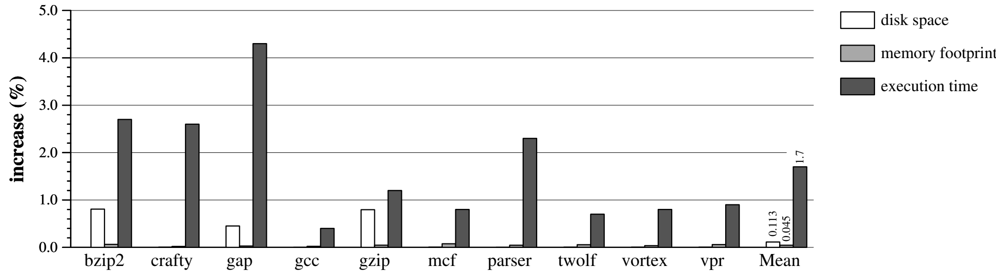


Figure 3: Time and space costs of IAT

system call verification mechanism, but due to a combination of increased page faults and a degradation in instruction cache performance.

NOP insertion incurs a runtime overhead of 5% on average, with two programs, *crafty* and *gcc*, incurring overheads of 8.5% and 9% respectively. This overhead comes directly from the increase in the number of instructions executed.

Layout randomization incurs a cost of 5.7% on average, with three programs experiencing significant increases in runtime: *crafty* (9.3%), *gcc* (14.7%), and *vortex* (10.2%). The cost increases here arise primarily from a degradation in instruction cache performance (see, e.g., Pettis and Hansen [25]). Our experiments indicate that unless layout randomization is done carefully, it can lead to a large increase in the number of TLB misses, resulting in a significant degradation in performance.

In comparison to other system call tracing based approaches such as Janus [16], Ostia [16], systrace [26] and ASC [29], the runtime overheads incurred are seen to be quite modest[3]. The worst case micro-benchmark overheads (35%) are a much smaller than other approaches (Janus: 10×, Ostia: 12×, systrace: 25× and ASC: 3×). Similarly, the worst case benchmark overhead (15%) are also quite comparable (Janus: 8%, Ostia: 25%, systrace: 30% and ASC: 3%).

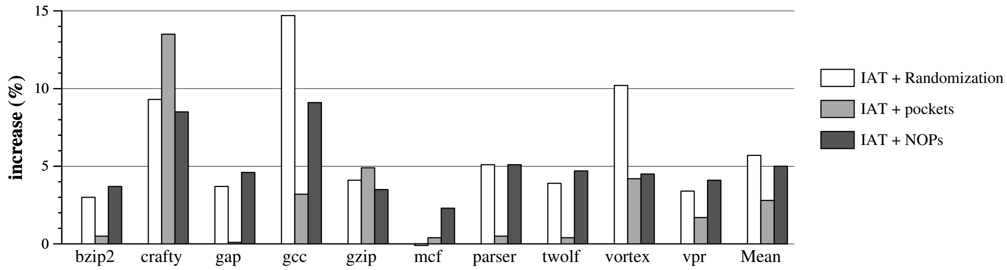
5.4.2 Space Cost

We considered two different aspects of space: the memory footprint of a program, and the amount of disk space it occupies. For each program, the disk space was obtained simply from the size of the executable file; its memory footprint was measured by examining the program header table and adding up the sizes of each segment that is to be loaded into memory (i.e., the PT_LOAD flag is set); in the case of pocket insertion, we then subtracted out the space occupied by pockets. In general, the disk and memory footprints of a program will be different, for two reasons. The first is that not all sections in the disk image of a program are placed in memory (e.g., the IAT section is not), while not all sections in memory are represented explicitly in the disk image (e.g., the *bss* section). The second is that the pocketing transformation introduces unused pages into the executable that affect its disk size but not its memory size. The increase in memory footprint size for our benchmarks is shown in Figure 4(b), with the effects on disk size shown in Figure 4(c).

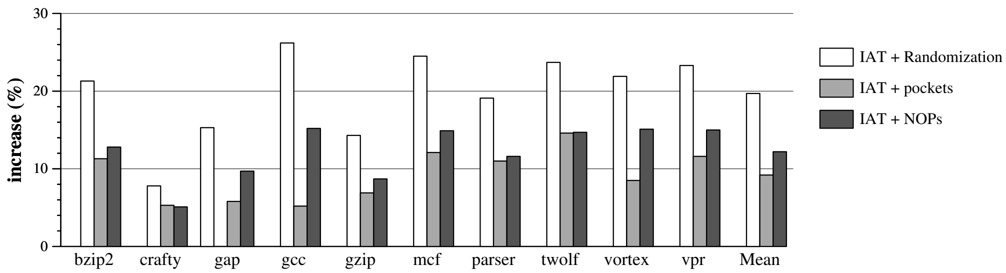
The increase increase in the memory requirements of a program due to the introduction of the IAT is minimal in user space and only approximately 8*n* bytes in kernel space, where *n* is the number of system calls in the program (the IAT has two 4-byte entries per system call). Since *n* is typically quite small in most programs, the memory impact of the IAT is also small. The bulk of the memory increases result from the secondary defenses, i.e., layout randomization, `nop`-equivalent insertion, and pocket insertion. On average, the overall memory cost is not large, ranging from about 9% for pocket insertion to 12% for NOP insertion, to about 20% for layout randomization. The largest increases are seen for layout randomization, where several benchmarks incur memory footprint increases of around 25% (e.g., *gcc*: 26.2%; *mcf*: 24.5%; *vortex*: 23.7%).

The increases in disk size are also reasonable for both layout randomization and NOP insertion, with overheads of 21.6% and 13.5% respectively. However, the space requirements for pocket insertion are much larger than the respective memory requirement (89.5% on average). This is due to the fact that while the actual insertion of pockets does not increase the memory footprint of the affected executable since these pockets are unmapped at

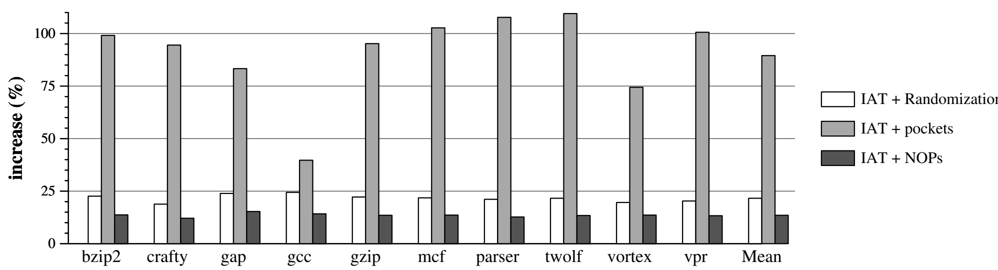
[3] Since each system provides different levels and types of security, a direct comparison is not possible.



(a) Execution Time



(b) Memory footprint



(c) Disk size

Figure 4: Time and space costs for thwarting mimicry attacks

runtime, the padding still takes up space in the file.[4]

6 Extensions and Future Work

The defence mechanisms described in this paper focus primarily on "control flow attacks," which try to manipulate the program's normal control flow in order to execute the appropriate system call(s). They currently do not address "data attacks" based on altering the data that get propagated in the system, for example, by changing the arguments to system calls. Such attacks are difficult to track since data values are generally not known until runtime and can potentially vary with each execution. Secondly, features such as interpretation, run-time code generation and templates, which are seen in many new programming models, introduce levels of indirection which can potentially be exploited to attack the system. In this section we discuss such attacks, and propose simple extensions which may be used to formulate appropriate defences.

6.1 Data Attacks

Argument Hijacking Attacks.
An *argument hijacking attack* is one in which the goal is to replace the arguments to a legitimate system call with those of an attackers choice. A simple defence would

[4]The pockets actually do contribute to the memory image initially, but are unmapped before execution of the original executable.

rely on extending code randomization to methodically hide system call parameters. Randomization techniques can be used to try and make it difficult for attack code to identify parameters, by making it appear as though all system calls have the same number of parameters (*system call homogenization*) and by randomizing the order of arguments (*argument randomization*) in a way that potentially allows each call site to use its own permutation [28]. Incorporating such a scheme would be straightforward from an implementation perspective and would require trivial modifications to the IAT. Specifically, for each entry in the table one would need to store two additional fields indicating correct parameter locations and the exact argument order.

A more sophisticated defence would be through the realization of a system call monitor [29]. To do this, one would need to modify the IAT so that each entry contains additional information which encodes for each system call argument a set of acceptable values. Thus each entry in the IAT would correspond to a call-site specific system call policy which can, at runtime, be checked each time the system call occurs. Techniques described by Rajagopalan *et al.* can directly be applied for automatically deriving system call policies and realizing more sophisticated monitors [29].

Interpreted programs.
A related problem arises with interpreters embedded in applications. Programs executed on such interpreters are viewed as data to the underlying system. This means that if a "bad" system call is executed in interpreted code—either because the injected attack code is interpreted, or because the attack code has modified part of an interpreted program—then this system call will be seen by the underlying defences as coming from a legitimate address, and will be allowed to execute. Such attacks are not addressed by the techniques proposed here.

6.2 Advanced Scanning Attacks

Templates.
Implementations of object oriented programming languages, such as C++, keep function pointers associated with the virtual methods in a class in a structure called a *vtable*, which is typically stored in static memory. Scanning the static memory region to locate vtables can be used as the starting point for a scanning attack. If a vtable is identified, the attack can potentially bypass protection mechanisms such as pocketing. One way to defend against this is using fine-grained pocketing, possibly by inserting pockets between basic blocks.

Stack inspection.
Another related attack is one in which the adversary examines the stack in search of addresses (such as function return addresses or function pointers stored in local variables) which can serve as a starting points for scanning attacks. Randomizing the order of local variables within activation records may prevent known address attacks, and encrypting pointers (as done by PointGuard [8]) can be used to prevent accesses to return addresses and function pointers. Note that for any of these advanced scanning attacks, the attacker must inject a significantly complex mechanism such as an interpreter or a simulator into the running program. A related side effect would be that programs under attack would show a drastic increase in their cycle count. This anamolous behaviour would potentially trigger an alarm in any intrusion detection system.

Dynamic Interfaces.
Distributed object technologies such as Microsoft COM suffer from the same vulnerabilities as object oriented languages, in that they place structures containing function pointers in memory. In COM, interfaces to objects are dynamically generated and passed in from outside the application program. Even if the application program was obfuscated, these external objects would likely not be obfuscated, and the interface structures could easily be scanned and identified by an attacker. Furthermore, COM identifies its objects and interfaces through Globally Unique Identifiers (GUIDs), and this makes it easy for an attacker to determine objects and derive information such as the type and the operations supported.

One solution that we propose would be the introduction of a small shim layer between the application and the COM infrastructure. Binary rewriting techniques could be used in the application program to encrypt the GUIDs and permute the function pointers in the COM interface structures. The shim layer would decrypt and de-permute these data structures before sending them to the COM infrastructure. Systems such as COM are significantly more complex and hence securing them is non-trivial, and an area of future work.

7 Related Work

The work that is closest to ours is that Rabek *et al.*, who propose monitoring the origin of library calls for the Windows operating system to prevent misuses of critical functions [27]. Their particular approach suffers mostly due to the fact that intercepting attack code at this level is vulnerable to mimicry attacks that "spoof" the return address on the stack. The approach can also be bypassed by the scanning attacks described here. Also related is the work of Bernaschi *et al.*, who propose modifications to the Linux operating system to regulate the usage of security-critical system calls [2]. System calls are intercepted at the kernel level and are validated based on rules stored in database. An example rule is validation of arguments known to be valid or safe. A drawback of this approach is that it requires manual encoding of access control rules for individual system calls and applications.

Du Varney *et al.* have proposed embedding semantic information into ELF binaries via an added section [9]. This work aims to simplify the task of post-processing executables for security purposes using binary rewriting tools. Because of this, the nature of the information embedded into binaries by Du Varney *et al.* is very different from ours.

Bhatkar *et al.* propose the use of address obfuscation to foil known-address attacks [3]. The idea is to randomize the base addresses of the stack, heap and code regions, and add gaps within stack frames and at the end of memory blocks requested by *malloc*. This technique is effective against known address attacks but is susceptible to the scanning attacks described in this paper.

There is a wide body of literature on defending against code injection attacks. Several researchers have proposed static program analysis to detect potential vulnerabilities such as buffer overflows [15, 20, 36]. When applied thoroughly, such schemes have the advantage of not letting an attacker even begin an attack. One disadvantage of such schemes is that they require that programs be recompiled using special compilers. This makes it difficult to apply them to third-party software, where the source code is unavailable and the conditions under which the binary was produced are not known.

ExecShield [34] prevents code injection via buffer overflow by making the process's heap and stack non-executable. This is difficult on the x86 architecture because it lacks separate "read" and "execute" page protection bits; ExecShield solves the problem by limiting the size of the code segment and putting the stack and heap beyond the end. Although this technique prevents code injection attacks, it does not prevent overwriting the return address with the address of a library, e.g. *system*. ExecShield performs address randomization to mitigate this type of attack, although this requires compiler support. ExecShield does not support programs that legitimately have executable content on the stack such as trampolines. These programs must have a flag set in the executable header indicating that ExecShield must not be invoked.

Other techniques, such as StackGuard [10] and FormatGuard [11], aim to prevent control transfers to the attack code. As in the previous case, such schemes require that programs be recompiled using special compilers, include files, and/or libraries, making them difficult to apply to third-party software. Moreover, they can be bypassed by well-crafted attacks (see, e.g., [4, 30]). There has been some recent work on disrupting the actual execution of attack code by means of "instruction set randomization" [1, 18], but current proposals for this have the drawback high execution overheads in the absence of specialized hardware support. Finally, Chew and Song have proposed techniques such as randomization of system call numbers [5]; a drawback of such approaches is its inflexibility in dealing with third-party software.

The idea of constructing semantic models of "legitimate" system call behaviors for a program in terms of sequences of system calls, and monitoring departures from such models, was proposed by Forrest *et al.* [13, 17, 38] and subsequently explored by a number of researchers (see, for example, [12, 19, 32, 35]). A drawback to this approach is that it is vulnerable to specific mimicry attacks [37]. Several of these schemes use the return address pushed by a system call to identify its call site [12, 32]. While this resembles our approach of identifying legitimate system calls based on the return address pushed by the software trap instruction, a significant difference between the two approaches is that our use of the IAT mechanism allows for other defenses against mimicry attacks, in partcular the system call cloaking scheme described in Section 4.1. Another difference is that schemes that rely on using training inputs to construct their semantic models of "good" executions have the drawback that it is difficult to ensure adequate code coverage, making for the possibility of false positives; by contrast, our approach is static, and so does not not suffer from runtime code coverage issues.

The use of NOP-insertion and code layout randomization to obfuscate code structure were proposed by Forrest *et al.* [14]; however, this work does not describe an implementation or provide experimental results. Other work along these lines is that of Wroblewski [39]. Many of these ideas can be traced to the Cohen's work on system diversification [6]. Additional techniques for binary obfuscation, to hamper static disassembly, are described by Linn and Debray [22].

Finally, several authors have proposed static analyses and/or type-based schemes to detect potential security vulnerabilities that could lead to the injection and activation of attack code [15, 20, 24]. Such schemes have the considerable merit of preventing the injection of attack code in the first place, which renders moot the issues addressed in this paper. A major drawback with such schemes is that they assume sufficient control of the code bases of all of the applications to be run on a system, so as to allow their analyses to be run on the source code. This is not always a realistic assumption in practice, since many applications are sold or distributed only as binaries.

8 Conclusions

Code injection attacks on software systems have become commonplace. Such attacks must eventually execute one or more system calls to cause damage outside of the compromised process. This paper describes a comprehensive approach for preventing the execution of such system calls. The core idea is twofold: first, use a table of addresses of "allowed" system call interrupt instructions to determine whether a given system call was executed from attack code; and second, use several different techniques to thwart mimicry attacks that attempt to get around this by identifying and executing system calls in the program code or in libraries. Our experiments indicate that the technique is effective and incurs only small runtime overheads. From a pragmatic perspective, it is also flexible: first, it is possible to run unmodified third-party software transparently, if desired, without any problems; and second, the additional information needed for our approach can be obtained using a binary rewriting approach on an executable, which means that it is not necessary to recompile the source code for an application using special compilers or libraries.

Acknowledgements

The work of Linn, Rajagopalan and Debray was supported in part by the National Science Foundation under grants EIA-0080123, CCR-0113633, and CNS-0410918. Discussions with R. Sekar and comments by the anonymous reviewers were very helpful in improving the contents of the paper.

References

[1] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 281–289, 2003.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *Proc. ACM Conference on Computer and Communications Security*, pages 174–183, 2000.

[3] S. Bhatkar, D. C. Du Varney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. 12th USENIX Security Symposium*, pages 105–120, 2003.

[4] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56), May 2000.

[5] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Electrical and Computer Engineering Department, Carnegie Mellon University, Pittsburgh, PA 15213, Dec. 2002.

[6] F. B. Cohen. Operating system protection through program evolution, 1992. `http://all.net/books/IP/evolve.html`.

[7] C. Collberg, J. H. Hartman, S. Babu, and S. K. Udupa. Slinky: Static linking reloaded. In *USENIX 2005 Annual Technical Conference*, pages 309–322, Apr. 2005.

[8] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. 7th. USENIX Security Symposium*, pages 63–78, Jan. 1998.

[9] D. Du Varney, S. Bhatkar, and V. Venkatakrishnan. SELF: a transparent security extension for ELF binaries. In *Proc. New Security Paradigms Workshop*, Aug. 2003.

[10] C. C. *et al.* StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th. USENIX Security Symposium*, pages 63–78, Jan. 1998.

[11] C. C. *et al.* FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[12] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE 2003 Symposium on Security and Privacy*, pages 62–77, May 11–14 2003.

[13] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *Proc. IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[14] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[15] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 345–354, 2003.

[16] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium*, February 2004.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[18] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 272–280, 2003.

[19] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proc. European Symposium on Research in Computer Security (ESORICS)*, volume 2808 of *Springer LNCS*, pages 326–343, 2003.

[20] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. 10th. USENIX Security Symposium*, pages 177–190, Aug. 2001.

[21] J. R. Levine. *Linkers and Loaders*. Morgan Kaufman Publishers, San Francisco, CA, 2000.

[22] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, pages 290–299, Oct. 2003.

[23] R. Muth and S. K. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL-00)*, pages 67–80, Jan. 2000.

[24] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. 29th ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Jan. 16–18, 2002.

[25] K. Pettis and R. C. Hansen. Profile-guided code positioning. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[26] N. Provos. Improving host security with system call policies. *Proceedings of the 12th USENIX Security Symposium*, 2003.

[27] J. C. Rabek, R. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *Proc. 2003 ACM Workshop on Rapid Malcode (WORM)*, pages 76–82, New York, N.Y., 2003. ACM Press.

[28] M. Rajagopalan, S. Baker, C. Linn, S. Debray, R. Schlichting, and J. Hartman. Signed system calls and hidden fingerprints. Technical report, TR04-15, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, May 2004.

[29] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated System Calls. In *Proc. IEEE International Conference on Dependable Systems and Networks (DSN-2005)*, June 2005.

[30] G. Richarte. Bypassing the StackShield and StackGuard protection: Four different tricks to bypass StackShield and StackGuard protection. Technical report, Core Security Technologies, Apr. 2000. `http://www2.corest.com/corelabs/papers/index.php`.

[31] B. Schwarz, S. K. Debray, and G. R. Andrews. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[32] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proc. IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[33] Symantec Corp. Understanding and managing polymorphic viruses. Technical report, 1996.

[34] A. van de Ven. New security enhancements in Red Hat Enterprise Linux. `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`.

[35] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[36] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed System Security Symposium*, pages 3–17, Feb. 2000.

[37] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detecion systems. In *Proc. 9th. ACM Conference on Computer and Communications Security (CCS)*, pages 255–264, 2002.

[38] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proc. IEEE Symposium on Security and Privacy*, 1999.

[39] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[40] T. Yetiser. Polymorphic viruses: Implementation, detection, and protection. Technical report, VDS Advanced Research Group, 1993. `http://www.virusview.net/info/virus/j&a/polymorf.html`.