A Multi-Faceted Defence Mechanism Against Code Injection Attacks

Department of Computer Science
University of Arizona
Tucson, AZ 85721.
Email: {linnc,mohan,bakers,collberg,debray,jhh,moseley}@cs.arizona.edu

Abstract

We propose a flexible host-based intrusion detection system against remote code injection attacks. 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 take advantage of 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, where 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 in recent years. 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. After this, 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). Finally, the execution of the attack code causes the various actions relating to the attack to be carried out.

It turns out that in general, 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, a number of researchers have focused on the system call interface as a convenient point for detecting and disrupting such attacks. For example, Chew and Song have proposed techniques such as permuting system call numbers in order to reduce the likelihood that the injected attack code will actually be able to execute the system call(s) it aims to execute [5]. A different approach, proposed by Forrest et al. [10, 13, 28] and subsequently the subject of a considerable body of research (see, for example, [15, 23, 25]), involves constructing semantic models of “legitimate” system call behaviors for a program in terms of sequences of system calls it may execute, and attempting to detect intrusions by monitoring departures from such models.

This paper describes a host-based defence mechanism against code injection attacks that, like the approaches described above, attempts to prevent attack code from executing system calls. There are two issues that need to be

*This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.
addressed: the primary defence mechanism, which addresses direct attacks; and a secondary defence mechanism that thwarts mimicry attacks that aim to bypass the primary defence [27]. To this end, we take a two-pronged approach. First, we disallow system calls from within the attack code itself. This is done by embedding semantic information into executables specifying the locations of legitimate system call instructions. When a system call occurs, the operating system kernel verifies that it is a legitimate call by checking the address from which the system call trapped into the kernel; system calls invoked from other locations signal an intrusion. Second, we take steps to deter mimicry attacks, where the attack code disguises its behavior to make it appear legitimate [27]. The primary defence mechanism, described above, allows system calls only from “known” locations, thereby precluding the injected attack code from making system calls directly. In order to bypass this, mimicry attacks must therefore rely on hijacking some existing system call (or existing code that eventually makes a system call) in the executing program. To thwart these, we use number of code transformations, including code obfuscation, pocket insertion, and code layout randomization, as well as a novel mechanism to disguise traps from an application into the OS kernel.

In order to be practical, it is desirable for the defence mechanism to work transparently with third-party software whose source code may not be available. We implement our scheme by using binary rewriting technology to add a new section, containing information about system call locations, into ELF binaries. The OS kernel checks system call addresses only if an executable contains this additional section. Our approach is, therefore, flexible: if an executable does not contain this section, this intrusion detection mechanism is not invoked. This makes it possible to run unmodified third-party software without any problems, while making it possible to protect 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. Section 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. Finally, in section 6 we summarize previous work related to intrusion detection and conclude in section 7.

2 Background

Our approach to intrusion detection, as well as the steps we take to defend against mimicry attacks, 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: Section 2.1 discusses the system call mechanism; Section 2.2 describes the structure of an ELF binary; and Section 2.3 sketches the dynamic linking process. Readers who are conversant with these topics may skip them. Section 2.4 describes the attack model we assume.

2.1 The System Call Mechanism

In most modern computer systems, certain interactions between an executing program and the underlying hardware, as well as other executing tasks, are placed behind a layer of abstraction in the OS kernel. The interface through which these operations can be requested by user level processes is the system call interface. The exact implementation of system calls varies slightly 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.

In order 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.
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 [17].

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 number of 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 which holds initialized data, a .rodata section which contains read-only data, a .bss section which contains uninitialized data, and a .got section which contains the global offset table (this is discussed more in Section 2.3).

For our purposes, it suffices to note that 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 actually run directly as statically linked binaries are. Instead they contain an extra interpreter section (.interp) which contains the name of an interpreter (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 its symbol tables, etc. The dynamic linker then scans the target binary’s .dynamic section for any shared libraries that the binary has dependencies on.
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 routine 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. When such a linked routine is first invoked, control is transferred, through the PLT and GOT, to the dynamic linker, which uses the name of the function (accessible via the PLT linkage used to invoke it) to identify the function in the shared library’s exported symbol table. Once this has been done, it patches the function’s entry point address into the executable’s GOT, cleans up the stack, and transfers control to the target function. Because the GOT entry for the function is patched with that function’s actual entry point by the dynamic linker, subsequent invocations of that function branch to the code for that function without reinvoking the dynamic linker.

2.4 The Attack Model

This paper focuses on remote code injection attacks. To this end, we assume that the actual application code being attacked—which, as discussed in later sections, will have been subjected to various randomizing transformations—is not available to the attacker for offline analysis or reverse engineering. The attacker may have access to other randomized versions of the same application, but they are likely to be different from the particular executable on the particular computer that is being attacked. In other words, we assume some level of inscrutability, in that that the attacker has no way to directly 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. We assume, however, that the attack code is capable of such analyses, and do not assume a priori bounds on the amount of time or space that the attack code may use for such analyses. We assume that the algorithms and even the source code for our algorithms are available to the attacker.

Finally, we make the common assumption that in order to do any damage to the system outside the compromised process, the attacker must make use of system calls.

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 from which the interrupt instruction for the system call was executed: if this is not any of the known locations from which the application can possibly trap into the kernel, then it must be coming from within attack code. This raises a number of issues, which we discuss in the remainder of this section and the next: (Section 3.1) how is the set of “allowed” interrupt 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 interrupt instruction that is part of the program code (or a dynamic library)?

3.1 Constructing Interrupt Address Tables

The list of addresses from which a program can generate system call interrupts into the kernel can be produced in a variety of ways. For example, it is entirely possible to have specialized compilers and linkers that generate this information as part of the process of generating an executable (dynamically linked libraries are discussed in Section 3.3). We take the simpler route of using post-link-time binary rewriting (our implementation currently uses the PLTO binary rewriting system for Intel x86 ELF executables [22]) to extract the relevant information for each such interrupt instruction in the executable.¹ We then add it to the ELF executable as a new section, the Interrupt Address Table (IAT), modify the associated headers in the ELF file appropriately, and write the file back out. The IAT section is an

¹This assumes that there are no “hidden” system calls in the binary, i.e., system call interrupts executed from dynamically generated code on the stack or heap, or in code that is dynamically decrypted and executed.
optional component of an ELF executable. This allows executables that do not have this information to be executed transparently, as discussed in Section 3.2, albeit without the protection of the mechanism described here.

The information in the IAT consists, at this time, of two values for each system call interrupt instruction found in the binary (this could be extended, in needed, to incorporate additional semantic information): (i) the address of the instruction immediately following the interrupt 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: in principle, the kernel could use the address pushed on the stack by the interrupt 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 interrupt 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 transparently, without any of the checks described in this paper; thus, third party software can, if desired, be run without any problems. The remainder of this discussion focuses exclusively on executables containing an IAT section.

There are two reasons for copying the IAT information into a kernel data structure instead of leaving it in the process’ address space. First, this prevents an attacker’s code from being able to modify the table. Second, placing this information in the kernel data structure makes it simpler and more efficient for the kernel to access it.

When a system call interrupt 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. While our current implementation uses a simple linear search, this check can be made quite efficient using hashing in a straightforward way. The system call is considered to be legitimate, and allowed to proceed, if the address is found in the IAT; otherwise, a possible intrusion is signalled.

3.3 Handling Dynamically Linked Libraries

Unlike addresses in executable binaries, addresses in shared object libraries, or dynamically linked libraries, are statically unknown, since the address at which the library will reside is not known until the dynamic linker maps it in. Furthermore, since the dynamic linker, and not the OS, is responsible for mapping the shared libraries, getting any information about the shared library integrated into the kernel is complicated. Below we some propose measures to address these issues.

Since the address at which a shared library will be mapped by the dynamic linker is not known a priori, the IAT section of a shared object file cannot specify absolute addresses. Instead, we modify the format of the IAT section for shared libraries to use file offsets. We can then either modify the dynamic linker, or write a link wrapper [17] to make a local copy of the IAT for the shared object and update the entry offsets to absolute addresses after mapping has taken place. Suppose, for the sake of simplicity, that this is accomplished this via modifying the linker (in practice, using a link wrapper is more convenient). Once the linker has finalized the memory addresses in its local copy of the IAT, it notifies the kernel that the kernel level IAT information for the process needs to be updated to include this new table. The kernel then merges the information from this table into its data structure. This is done during the initial set up operations performed by the dynamic linker for each dynamically linked library referenced by an executable, when these libraries are being mapped into the process’s address space.

To allow the dynamic linker to notify the kernel of additional IAT sections to be merged in, we add a new system call to the kernel. The arguments to this system call are the base address and size of an IAT to be added to the current process’s IAT information. Using the dynamic linker to merge in the IAT information for the shared libraries in this

2Because plto [22], the binary rewriting system we used for our implementation, handles only statically linked binaries, this portion of our work has not been implemented, though in principle it is not difficult to do so.
way, the addresses of all system call interrupts in the shared libraries being loaded will appear in the kernel level IAT for the process before the program actually begins to execute. 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 turns out not to be a problem since the kernel is responsible for mapping the interpreter into the executable (before the process begins execution), hence it can retrieve and patch the dynamic linker’s IAT section much in the same way the dynamic linker handles other shared objects, and finally add it into the kernel level verification table before control is ever passed to the interpreter.

It turns out that one last modification to the dynamic linking process is needed, to deter mimicry attacks: specifically, the default lazy binding mechanism used to resolve dynamically linked routines. This is discussed in more detail 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 [27]. This allows such attacks to bypass intrusion detection systems that focus on detecting anomalous behaviors.

Using IAT information, as described in the previous section, it is possible to identify any system call interrupt that is in the injected attacked code, since the addresses for such instructions will not appear in the IAT. The only way to get around this is for the the attack code to use a system call interrupt instruction that is already in the program: either part of the program code, or in some shared library. This section discusses the forms such attacks take and the steps we take to address such attacks.

In order to use a system call interrupt instruction that is part of the program, the attack code must branch either (i) to the interrupt instruction itself, or (ii) to some location from which execution eventually reaches such an interrupt instruction, e.g., some function in the standard C library. There are two possibilities here. The first is that of a “known address attack,” where the attack code jumps to some fixed address that contains (or leads to) a system call interrupt 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 interrupt 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 interrupt 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 interrupt 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 Interrupts

One weakness in existing executables is that system call interrupts are easily identifiable, making them potentially vulnerable to scanning attacks, as described above. We can address this by making system call interrupts 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 figure out whether the trap is actually a system call interrupt in disguise.

Recall that using the IAT, the kernel has enough information to identify whether a particular trap into the kernel was executed at an address corresponding to that of a legitimate system call interrupt instruction. Since all traps into the kernel are handled in pretty much the same manner, it is not difficult to implement a scheme where the true nature of an interrupt is ambiguous until the kernel disambiguates it using the IAT information for the process. The kernel is
now modified so that any trap into the kernel is first checked against the process’s IAT: if the address of the instruction
generating the trap is thereby found to correspond to a system call interrupt, it is processed as a system call; otherwise,
it is processed as a normal trap.

This scheme can be quite effective in disguising system call interrupt 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 interrupt in an application or library can be varied randomly across different
systems.

4.2 Hindering Scanning Attacks

Once system call interrupt 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 a number of 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 sym-
bol 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 rou-
tines (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 staying with lazy binding for dynamically linked
routines, since the standard lazy binding mechanism relies on the availability of symbol information, which in turn
opens up the possibility of scanning attacks that look up the symbol table. 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,
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.

4.2.2 Dead and Useless Code Insertion

A simple way to disrupt attacks that scan for specific byte sequences is to periodically insert (randomly chosen) instruction sequences into the text stream that do not alter program semantics, but which change the byte sequence for the program text [11]. 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 [24, 30]. The approach can be enhanced using binary obfuscation techniques [18].

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 [11]. 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 [18], 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 non-contiguous segments, separated by “pockets” of invalid addresses. If the attack code ever hits 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 unmapping 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

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.

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 a variant of plto [22], a general purpose binary rewriting tool for the Intel IA-32 executables. Plto 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.

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.

5.1 Efficacy

To test the efficacy of our implementation for our attack model we constructed a series of synthetic attacks on a sample program taken from the SPEC-95 benchmark suite: m88ksim, a simulator for the Motorola 88000 processor. 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). Since our aim is to prevent an attacker from executing system calls, we constructed a suite of synthetic attacks that attempted to execute system calls using a number of different attack scenarios, including both direct invocation of system calls using an ‘int 0x80’ instruction in the attack code, and mimicry attacks involving scanning. Since our research is not concerned with preventing attack code from being injected into an executing program, our synthetic attacks simply assumed that the attack code had somehow been introduced into the application and executed: we simulated the direct attack by injecting the attack code onto the runtime stack and branching to it; the mimicry attacks were written in C and linked in as part of the program.

As discussed below, our approach was successfully able to detect each attack.

5.1.1 Injected System Call Interrupts

The first class of attacks we considered represent attacks that attempt to execute a system call interrupt instruction directly from the injected attack code. In practice, such an attack might result from overflowing executable code onto the stack and executing it in place, or overflowing attack code onto the heap, etc. Our test that represents this sort of attack was modeled using a simulated buffer overflow attack, where instructions were first pushed onto the stack, then executed by jumping into the code on the stack.

Our tests show that this class of 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 is correctly declares the interrupt malicious and take appropriate action.
5.1.2 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. Bhatkar et al. have demonstrated the efficacy of such randomization techniques against “known-address attacks,” where the attack code branches to some fixed address hard-wired into it [3]. We therefore did not separately examine known-address attacks in our experiments.

5.1.3 Scanning Attacks

We examined a number of scanning attacks that used pattern matching to try and discover the locations of valid system call entry points. We considered three levels of such scanning attacks: (i) those scanning for any existing system call interrupt (int 0x80) instruction; (ii) those scanning for an interrupt with a specific system call number; and (iii) those scanning for a known library entry point by pattern-matching against a byte sequence signature.

Each synthetic attack, at each of these levels, was unsuccessful against the implemented intrusion detection measures, namely use of the IAT in combination individually with each of nop-equivalent insertion, pocket insertion, and layout randomization.

5.2 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.

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>
<thead>
<tr>
<th>System Call</th>
<th>Time w/o IAT (µsec)</th>
<th>Time with IAT (µsec)</th>
<th>% Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>getpid</td>
<td>0.71</td>
<td>0.96</td>
<td>35.2</td>
</tr>
<tr>
<td>open</td>
<td>19.58</td>
<td>19.77</td>
<td>1.0</td>
</tr>
<tr>
<td>read</td>
<td>95.75</td>
<td>98.19</td>
<td>2.5</td>
</tr>
</tbody>
</table>
```

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

3 We were unable to build two other benchmarks in the suite, perlbmk and eon.
<table>
<thead>
<tr>
<th>Program</th>
<th>Disk space (bytes)</th>
<th>Memory size (bytes)</th>
<th>Execution time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>IAT</td>
<td>IAT/Orig</td>
</tr>
<tr>
<td>bzip2</td>
<td>508246</td>
<td>512350</td>
<td>1.00807</td>
</tr>
<tr>
<td>crafty</td>
<td>676200</td>
<td>676208</td>
<td>1.00001</td>
</tr>
<tr>
<td>gap</td>
<td>909674</td>
<td>913778</td>
<td>1.00451</td>
</tr>
<tr>
<td>gcc</td>
<td>1876310</td>
<td>1872222</td>
<td>0.99782</td>
</tr>
<tr>
<td>gzip</td>
<td>516438</td>
<td>520542</td>
<td>1.00795</td>
</tr>
<tr>
<td>mcf</td>
<td>450902</td>
<td>450910</td>
<td>1.00002</td>
</tr>
<tr>
<td>parser</td>
<td>581974</td>
<td>577886</td>
<td>0.99298</td>
</tr>
<tr>
<td>twolf</td>
<td>643414</td>
<td>643422</td>
<td>1.00001</td>
</tr>
<tr>
<td>vortex</td>
<td>1024342</td>
<td>1024350</td>
<td>1.00001</td>
</tr>
<tr>
<td>vpr</td>
<td>586070</td>
<td>586078</td>
<td>1.00001</td>
</tr>
<tr>
<td>Geom. mean</td>
<td>1.00113</td>
<td>1.00045</td>
<td>1.017</td>
</tr>
</tbody>
</table>

Table 1: Cost of Instruction Address Tables

5.3 Cost of Transformations to Thwart Mimicry attacks

5.3.1 Time Cost

The effect of using various techniques for thwarting mimicry attacks on execution time is shown in Figure 2(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 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., [19]). 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.

Overall, the runtime overheads incurred due to the various measures to thwart mimicry attacks are seen to be quite modest.

5.3.2 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 its program header table and adding up the sizes of each segment that is to be loaded into memory (i.e., has its PT_LOAD flag 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 2(b), with the effects on disk size shown in Figure 2(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 8n 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
Figure 2: Time and space costs for thwarting mimicry attacks

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 runtime, the
6 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 [20]. 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. A number of researchers have proposed static program analyses to detect potential vulnerabilities such as buffer overflows [12, 16, 26]. 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. Other proposals, such as StackGuard [8] and FormatGuard [7], 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, 21]). There has been some recent work on disrupting the actual execution of attack code by means of “instruction set randomization” [1, 14], 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 to reduce the likelihood of the injected attack code working the way it is supposed to [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. [10, 13, 28] and subsequently explored by a number of researchers (see, for example, [15, 23, 25]). A drawback to this approach is that it is vulnerable to specific mimicry attacks [27].

The use of NOP-insertion and code layout randomization to obfuscate code structure were proposed by Forrest et al. [11]; however, this work does not describe an implementation or provide experimental results. Other work along these lines is that of Wroblewski [29]. 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 [18].

7 Conclusions

Code injection attacks on software systems have become commonplace in recent years. In order to affict harm, such attacks must eventually execute one or more system calls. This paper describes a multi-faceted approach for preventing the execution of such system calls. The core idea is twofold: first, use a table of addresses of “allowed” system call

---

4The pockets actually do contribute to the memory image initially, but are unmapped before execution of the original executable.
interrupt instructions to determine whether a given system call was executed from attack code; and second, use a number of 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.

References


http://all.net/books/IP/evolve.html.


