Signed System Calls and Hidden Fingerprints *

M. Rajagopalan,† S. Baker,† C. Linn,† S. Debray,† M. Hiltunen,‡ R. Schlichting,‡ J. Hartman†

† Department of Computer Science
The University of Arizona
Tucson, AZ 85750

‡ AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932

Abstract

Remote code injection attacks against computer systems are occurring at an alarming frequency. A crucial aspect of such attacks is that in order to do any real damage, the injected attack code has to execute system calls, and therefore can be foiled by suitably hardening the system call interface. Most current proposals for doing so, however, suffer from various shortcomings, such as relying on special compilers or libraries, or incurring huge runtime overheads, or being vulnerable to mimicry attacks. This paper describes a systematic approach to defending against remote code injection attacks that uses two complementary techniques: cryptographic signatures to protect system calls themselves, and compiler-based techniques to hide code fingerprints that could be exploited for mimicry attacks. Experiments indicate that our approach is effective against a wide variety of attacks at modest cost.

1 Introduction

Worms, viruses, denial of service attacks, and other security incidents are occurring at an alarming frequency despite the increased attention being paid to computer security [13]. These attacks utilize a variety of system vulnerabilities ranging from careless users (e.g., clicking on executable email attachments), to lax system administration (e.g., default, weak, or non-existent passwords), to vulnerabilities in the system or application programs (e.g., buffer overflows). While compromising even a single machine can cause significant damage, the largest overall impacts are typically caused by *self-propagating attacks* (such as the Slammer, Sasser, or Nimda worms) that exploit a common vulnerability to automatically spread to a large number of machines. Such attacks are often based on code injection into a running program. In a typical attack scenario, a remote attacker exploits some software vulnerability, such as lack of protection against buffer overflows, to introduce attack code into the system. The system is then "tricked" into executing this code, thereby allowing the attacker to obtain control of the application.

Taking control of an application is, however, not enough to be useful to an attacker. In order to do any real damage—such as create a root shell, install or change permissions on a file, or access proscribed data—the attack code needs to interact with other parts of the system by executing system calls. If the attack code can be prevented from doing this, the intruder will not be able to gain control over the machine. The system call interface is therefore a crucial defense point against intrusions.

In this paper, we describe a systematic defense against code injection attacks that uses a combination of cryptographic and compiler-based methods to harden the system call interface. Specifically, we use two complementary techniques, *system call signing* and *fingerprint hiding*, to prevent attack code from executing the system calls it requires to gain control. Our approach is based on using an *installer* program to transform the executables of applications and user-level system software such as daemons before execution. Specifically, the installer performs two types of transformations. The first is to modify each system call to include a signature constructed using standard cryptographic techniques, thereby creating a *signed system call*. This signature is then used by the kernel at runtime to verify that the system call was not created or modified by malicious code. The second

*This work was supported in part by the National Science Foundation under grants EIA-0080123 and CCR-0113633.

transformation hides *fingerprints*—that is, the binary patterns associated with specific code segments—to prevent attack code from using such patterns to locate and exploit existing system call stubs and functions that make system calls. It does this by randomly relocating and modifying a program's basic blocks using correctness-preserving compiler transformations.

Our approach has many features that make it practical and effective. One is that it is fully automated, requiring only that a system administrator run the installer for each application and service that executes on the machine. It also does not require access to the source code, and works for both statically and dynamically linked executables. Another advantage is that it does not require keeping the algorithms or the approach itself secret, since the signing security is based on cryptographic techniques and the fingerprint hiding is based on randomization. The random nature of fingerprint hiding techniques makes it nontrivial to detect the different types of variations introduced in programs. The overhead of the approach is moderate (see section 4), depending primarily on the strength of the cryptography used, and is mostly incurred at installation time rather than at run time. Finally, the approach not only prevents code injection attacks, it also makes it more difficult for an attacker to install new software (e.g., backdoors, password sniffers, DDoS bots[1]) on an otherwise compromised machine.

While other research efforts have addressed buffer overflow attacks and the system call interface, none provides techniques as effective, efficient, and well-founded in cryptography and compiler techniques. Most of this research has focused on ways to prevent code injection attacks from happening, with proposals ranging from techniques for detecting potential buffer overflows [20, 26, 27], to approaches for preventing the execution of attack code [6, 16, 17], to dynamic decryption techniques to disrupt the execution of the attack code [8, 24]. The first two approaches have the drawback of requiring that applications be recompiled using special compilers, header files, and/or libraries, which makes it difficult to apply them to third-party software whose source code may not be available (they may also be vulnerable to carefully crafted attacks [11, 31]). The third approach incurs extremely high performance overheads on stock hardware. Recently, the system call behavior of programs has received considerable attention in the context of intrusion detection [18, 23, 25, 34, 37, 39, 40]. The main drawback of such systems has been their reliance on maintaining application specific state in the kernel.

[1] DDoS (distributed denial of service) bots, or DDoS zombies, are programs that are installed on compromised machines and that launch DDoS attacks against a given target on command.

This paper focuses on remote code injection attacks. We assume that our algorithms are known to the attacker, but not the secret keys used to protect applications on a particular system. We also assume that the actual application executable on the machine being attacked is not available to the attacker for offline analysis or reverse engineering. The attacker may have access to the source code or other transformed versions of the same application, but—because of randomizing transformations—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 the attacker has no way to directly determine the instruction sequence or layout of the code being attacked.

The rest of this paper is organized as follows. Section 2 explains our approach in detail and describes how it prevents the different types of code injection attacks. The implementation details of the approach are described in section 3. We currently have two prototype implementations of the approach, one using Linux kernel modifications and the PLTO binary re-writing tool, and another using an unmodified version of Linux running on the Xen virtual machine monitor [7]. Section 4 provides preliminary experimental results of the approach including the performance overhead and evaluation of the effectiveness of the approach. We describe the related work in more detail in section 5 and section 6 summarizes the contributions of the paper and outlines future work.

2 Our Approach

Our approach is based on transforming binary program executables by applying the techniques of system call signing and code fingerprint hiding. These transformations are performed by an installer program that is executed only by a system administrator authorized to install software on the particular computer. For system call signing, the administrator provides the installer a secret key used for cryptographic functions in the installer. The installer is a binary rewriting tool that reads in the binary executable of the program to be installed, disassembles it, and constructs a control flow graph for the program. The installer then locates the system calls in the binary, for example, by scanning the code of the binary for occurrences of the '`int 0x80`' system call interrupt instruction. Finally, it computes the signature at each system call in the program, modifies the system call to pass this signature to the kernel, transforms the control flow of the program to hide fingerprints, and then writes the transformed binary back to disk. The system call signing transformation is described in detail in Section 2.1 and the fingerprint hiding transformations are described in Section 2.2.

2.1 Signed System Calls

The key idea behind system call signing is to associate additional information, a *signature*, with a system call, in a manner that is difficult for an attacker to forge. The signature can then be checked within the OS kernel to verify that it is a legitimate system call from a properly installed program. The signature is constructed, and verified, using a cryptographic function and a secret key. Depending on the type of cryptographic function used, the key may be shared by the kernel and the program that constructs the signature (the installer program) or the key used by the kernel to verify a signature may be publicly known while the key used to construct the signature is secret (public key cryptography [32]). In either case, the secret key used to construct the signature is never stored in the application program, where it is potentially accessible to attack code. Rather, the secret key is only provided to the installer program when it is run, and the installer uses the key at installation time to construct the signatures.

2.1.1 Constructing System Call Signatures

Cryptographically signed components of a system call are difficult for attack code to alter. For example, if (the string address of) the file name being passed to an `open` system call is part of its signature, then any attempt by attack code to change that argument, to point to a different string, will be detected by the kernel when it checks the signature. Intuitively, therefore, the larger the set of values encompassed by a signature, the fewer openings there are for attack code to exploit. More formally, the *coverage* of the signature can be defined as the metric of how much of the system call is protected by the signature. It is therefore natural to try and maximize the extent of coverage achieved. Ideally, the signature should protect the system call number, the address of the system call,[2] and all the system call parameters (passed in registers or the stack). If a signature covers the location of a system call, the attack code cannot reuse that signature for another system call at a different location. For the system call arguments whose values are known, e.g., using standard program analysis such as constant propagation , the signature can protect either the value of the argument, or—for address arguments—the dereferenced value, i.e., the value pointed at (e.g., the character sequence for the string `"/bin/csh"`). If the parameter address is covered, the attack code cannot replace the argument with one at a different address. If a parameter value is covered by the signature, the attack code cannot modify or replace that argument with a different value.

While signing the system call number is straightforward, there are more subtle issues with signing the location and argument values of a system call, involving a tradeoff of code size vs. extent of coverage obtained. For example, while the system call address is known and can be signed at installation time for statically linked binaries, the address is not known for dynamic libraries. For a program for which we want increased protection, therefore, we can create a statically linked executable that contains all the libraries that it requires: this results in a larger executable that is less vulnerable to attacks. We have a similar size/security tradeoff for system calls (or library routines) called from multiple call sites with different constant arguments. In such situations, we can inline the system call (or library routine) into each call site, thereby generating specialized versions of the code, each having its own set of signed parameters (the parameters to functions in dynamic libraries are typically not known at installation time, so they cannot be signed).

As this discussion indicates, a system call signature has to convey information about those aspects of a system call that are known at program install time; the information necessary to allow the kernel to retrieve the arguments from the appropriate locations; and, for each argument, whether or not the argument value is signed. A signature therefore consists of the following information:

- *system call number:* the original system call number;
- *argument order:* the order in which the actual system call arguments are passed to the kernel (as discussed in section 2.2, the installer randomly permutes the argument order at each system call location);
- *signature coverage:* describes which arguments of the system call have been covered by the signature and if the call address is covered by the signature; and
- *argument and system call address signature:* A CRC over the system call address (if known) and the argument values/addresses (if known). If neither is known, this field is filled with random bits.

This information is encoded into a bit sequence that is then encrypted using a cryptographic algorithm and a secret key. In our current implementation this information

[2] By the "address of a system call" we mean the address of the instruction following the '`int 0x80`' interrupt instruction. Since this address is pushed onto the stack by the hardware as a side effect of executing the system call interrupt, i.e., 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.

is encoded in 64 bits. Note that the system call number itself is included in the signature and is not passed to the kernel in the clear. The main reason for hiding the system call in this manner is to hide the fingerprint associated with this system call stub. The argument order is also related to fingerprint hiding, see Section 2.2. The fact that the system call number is included in the signature also allows us to use the register that would have passed the system call number to the kernel to store part of the signature instead.

2.1.2 Using System Call Signatures

When the operating system kernel receives a signed system call, it decrypts the signature using its key. It then verifies the signature by making sure the decrypted information consists of a valid system call number and syntactically valid argument order and signature coverage descriptions. It then uses these descriptions to calculate a CRC over the covered address and the arguments, and compares this to the one included with the signature. Only if everything matches does the kernel execute the system call. If the signature does not match, the kernel can be configured to perform a choice of actions including logging the event, alerting the operator, and killing the process that issued the incorrectly signed system call by executing the *exit* system call.

2.2 Hiding Code Fingerprints

While signed system calls prevent injected attack code from constructing a system call and invoking it directly, it may be possible for a sophisticated attack to indirectly invoke a system call already in the program, e.g., by finding and jumping to the beginning of a suitable library function. The *fingerprint* of such a function is any characteristic property of the function that the injected code can use to locate it; for example, the address of the function, if known to the attacker, is a trivial fingerprint.

To launch an indirect attack, the attack code can simply jump to some fixed address where it expects to find code that will lead to the desired system call (a *known-address attack*). Alternatively, it can use pattern matching with specific instruction sequences to identify code, such as library routines or system call stubs, that will eventually lead to the desired system call (a *scanning attack*). Such attacks rely on the program being attacked being predictable in some way: either having a particular routine at a predictable address, or having some predictable byte sequence that can be used to identify some routine. These attacks can therefore be handled by randomizing the structure of the programs in such a way as to destroy such predictability. This can be done using two basic techniques: code layout randomization and elimination of distinctive byte sequences within the code.

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 (in the latter case, it may obviously be necessary to add additional control transfer instructions to preserve program semantics) [19]. 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 encountered today. Moreover, such reverse engineering by the attack code can be thwarted using binary obfuscation techniques [28], which inject "junk bytes" into an executable to make disassembly algorithms produce incorrect results.

We use two complimentary techniques for eliminating distinctive byte sequences. For the system call stubs, we use *system call homogenization* and *argument randomization*. For all basic blocks, we break code fingerprints by using *random code insertion*. All system calls are homogenized by adding dummy arguments to system call, so that all system calls appear to take the same number of arguments.[3] This is done since information about the number of arguments can, in principle, be used by attack code to help identify specific system calls. As the next step, the order of arguments is randomized. This may cause, for example, the filename argument to an `open` system call to be passed, say, in the fourth argument position. Each system call location in a program has its own argument permutation, which means that two different calls to `open` in the same program could pass the arguments in different orders. The information about the actual argument order is passed to the kernel as part of the signature.

Random code insertion disrupts attacks that scan for specific byte sequences by periodically inserting into the text stream (randomly chosen) instruction sequences that do not alter program semantics, but which change the byte sequence for the program text [19]. 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

[3] The maximum number of arguments taken by any system call in Linux is 6, so system call homogenization makes every system call appear to take six arguments in our implementation.

to ensure that none of the condition codes affected by the inserted instructions is live at the point of insertion. The approach can be enhanced using binary obfuscation techniques [28]. 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 identify, for each function, the shortest byte sequence needed to uniquely identify that function in that program; and thereby determine 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. 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 [35, 44].

3 Implementation Approach

This section describes our implementation of a prototype system to evaluate the efficiency of the techniques.

3.1 Securing Application Binaries

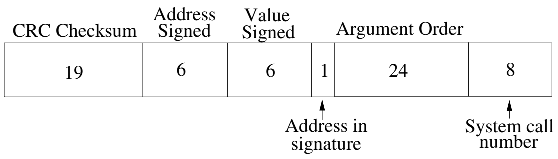

Figure 1: System Call Signature

In our prototype implementation, a system call signature consists of a 64-bit sequence as shown in Figure 1. It encodes the system call number, argument order, signature coverage and a CRC over the various fields making up the coverage. The bit sequence is encrypted using the DES algorithm using a key shared between the installer and the OS kernel. The signature is placed in memory, and the system call passes its address to the kernel in the `eax` register.[4]

We used the PLTO binary rewriting toolkit [33] to perform user level transformations for protecting applications. It uses constant propagation to identify the system call number for each system call. The implementation adds dummy operations where necessary, so that all system calls have the same number of arguments, then randomly permutes the arguments of each system call. The permutation order is recorded in the signature, so that the kernel can recover the arguments, in the right order, after it decrypts the signature. Our current implementation does not perform address and value signing of system call arguments, which we leave as a future extension.

Our tool also carries out various semantics-preserving transformations to hide code fingerprints (see Section 2.2). In particular, it inserts randomly selected instruction sequences that are semantically equivalent to `nops` into the instruction stream roughly k instructions apart, where k is slightly less than the average number of instructions per block; currently, these `nop`-sequences are chosen to be between 1 and 4 instructions in length. It also performs code layout randomization, randomly permuting both the basic block layout within functions as well as the order of functions within an executable. Section 4 demonstrates that the performance overhead due to these transformations is modest.

Our current implementation handles only statically linked binaries. This is not due to any limitations of our approach, but because of restrictions in the underlying binary rewriting tool PLTO [33]. We are working on extending PLTO to handle dynamically linked executables. After PLTO has been extended, our approach can be used directly to support dynamic linking: each library is signed and randomized through the installer, as described earlier. In addition, to prevent indirect and known address attacks that utilize features of dynamic libraries, we will randomize the library entry points and sections (GOT and PLT) of the ELF binary, and unmap portions of the symbol table. The exact details of these procedures are beyond the scope of this paper. Note, as mentioned earlier, that dynamic linking reduces the coverage of the signature since items such as return addresses can no longer be included in the signature.

3.2 Handling Signed System Calls

It is necessary for the operating system to decrypt system call signatures and use them to process the system call arguments. If source code is available for the OS kernel, then one alternative is to modify the kernel source to handle system call signatures. This has the advantage of having lower performance overhead and is ideal for target environments such a small mobile devices (e.g., cell phones) where system overheads are a premium for performance, power etc. Alternatively, kernel modifications can be avoided by handling the signed system calls in a virtual machine before they are passed to the kernel. This also has the advantage of supporting multiple underlying operating systems simultaneously (e.g., Windows

[4]Normally the `eax` register is used to pass the system call number. In our case, the system call number is part of the signature, and so does not have to be passed separately.

and Linux). We experimented with both approaches, as described below.

3.2.1 In-kernel interception

Interception in the kernel is done by modifying the software trap handler. Typically the software trap handler is responsible for identifying the system call number, based on the contents of the `eax` register, invoking the appropriate system call handler and returning the result to the calling application. We modified the handler to call a routine that decrypts the signature using DES and verifies the signature. If the checks succeed, the parameters are rearranged according to the argument order information passed as part of the signature, after which control is passed to the original system call handler identified through the original system call number. In case the checks fail, the process is terminated gracefully by invoking the `exit()` system call.

Interception in the kernel involved modifying the `entry.s` file and adding about 200 lines of source code for the signature decryption and verification routines.

3.2.2 Interception through a VMM

It may not always be feasible or practical to modify an operating system kernel to support signed system calls. In this case signed system calls can be implemented through a virtual machine monitor (VMM) without modifying the kernel. A VMM provides a virtual machine abstraction to the kernel, fooling it into thinking it is running on the bare hardware when in fact it is not. The basic idea behind implementing the signed system calls in the VMM is that when an application issues a system call, it causes a system call trap on the real hardware. This trap is caught by the VMM, which then invokes a virtual system call trap in the virtual machine. The operating system running in the virtual machine catches this trap just as if it occurred on the real hardware.

The signed system call functionality can be implemented without modifying the kernel by placing it in the system call trap handler in the VMM. When the VMM gets the trap, it invokes the routine to verify the system call signature. If the checks succeed, the parameters are modified as necessary and control passed to the operating system in the virtual machine. The operating system then processes the system call without any knowledge of signatures. If the checks fail, then the offending process receives a general protection fault, causing the process to be terminated.

We implemented a prototype of this in the Xen [7] VMM running an unmodified version of Linux. This required less than 50 lines of code to be added to the Xen system call handler. To simplify introducing code into Xen, we currently turn off Xen's *fast trap* mechanism, which speeds up system calls. This results in somewhat higher overheads for signed system calls in this implementation.

3.3 Signature Caching

To mitigate the cost of in-kernel decryption of system call signatures on mobile systems, such as cell-phones, where performance and power are a concern, we can add a software cache in the kernel. Each cache line contains the 3-tuple ⟨return address, signature, decrypted signature⟩. The idea is to use the return address of a system call as the index into the cache. Each time a system call is made, the corresponding entry in the cache is checked to see if the return address and the signature matched. If they do, the cached decrypted signature is returned; otherwise decryption is carried out as usual and the previous entry is overwritten with the new value.

As shown in Section 4, signature caching has the effect of amortizing the cost of in-kernel decryption over a large number of system calls, and thereby dramatically reduces the average cost of a signed system call. These savings can be attributed to the fact that programs typically contain a few system calls that are repeatedly called from the same location.

Our prototype implementation uses a 100-line cache. The source code for maintaining the cache is fewer than 40 lines long.

4 Experimental Evaluation

This section describes experiments performed to evaluate the effect of these transformations on a 3.2 GHz Pentium IV system with 1 GB of main memory running *Fedora Core 1* with a Linux 2.4.22 kernel. The experiments are described in two sets, the first aimed at testing the increased resilience to malcode injection through known attack techniques. The second set demonstrates the performance overheads imposed by these techniques.

4.1 Attack Experiments

Broadly speaking, a remote code injection attack begins by exploiting a software vulnerability to inject and execute some attack code. The execution of the attack code eventually results in (or is intended to result in) the execution of an appropriate system call. Most reported attacks differ in the details of the specific vulnerability that was exploited to inject the attack code, and/or the particular actions that were carried out once the attack gained control of a system. Because the focus of our work is on

Type of attack	*Attack Outcome*	
	Regular System	*Protected System*
Simple Code injection	attack succeeded (shell access)	attack failed (process exited without shell)
Jump to known address	attack succeeded (shell access)	attack failed (segmentation fault)
Fingerprint based scanning attack	attack succeeded (shell access)	attack failed (unable to find pattern)
Hijacking system call parameters	attack succeeded (shell access)	attack failed (process exited without shell)

Table 1: Attack experiments and their Outcomes

hardening the system call interface between an application and the OS kernel, neither of these facets of attacks is relevant to this paper: we focus, instead, on the manner in which the attack code causes a system call to be executed. Furthermore, we would like to be able to explore attacks that may not even have occurred "in the wild." For these two reasons, we used synthetic attacks to evaluate our ideas. We used four different synthetic attacks, each of them examining a different way to execute a system call, and each therefore representative of a whole class of "real" attacks. The mechanisms incorporated into our synthetic attacks were based on exploits from [3, 4, 5], viruses [2] and worms [1, 36]. The attacks were targeted at a synthetic program which contained several exploitable features such as an overflowable buffer and direct calls to the execve function.

Simple Code Injection
The simplest attacks involved injecting malicious code into the stack or heap of an executing program and then executing the malicious code, which invoked a system call directly. Most reported code injection attacks fall into this category [3, 4, 5].

Our experiment was based on injecting exploit code through a known buffer overflow in the vulnerable program (note that the nature of the exploit used to inject and execute the attack code—e.g., buffer overflow, heap overflow, double free, format string vulnerability, etc.—is not important for the purposes of these experiments). The attack was able to compromise the unmodified system but failed on the protected system. Code injection was successfully carried out even in the protected system, but when the malicious code tried to invoke a signed system call, this was detected and the process terminated.

In case of long running processes such as web servers, restarting the process as described in [12] would be an alternative to terminating the process.

Jumping to known address
Knowing that the malicious code can not make system calls directly, an attacker could try an jump to the address of a known system call. Since source code is available to the attacker – system call locations can be identified easily by disassembling the compiled program. This allows the attacker to exploit the uniformity shown in general systems and predict with a high probability the addresses at which functions exist. A number of platform neutral distributions of exploits demonstrate this observation.

Our attack experiment was based on identifying the location of the `execve` system call and jumping to it. In the unmodified system, the attack succeeded in compromising the system. In the protected system, the buffer overflow succeeded in injecting attack code, but because code addresses had changed due to randomization of the binary's layout, the injected code jumps to the wrong address, causing the attack to fail.

Scanning for signatures
The next set of attacks model function fingerprint based attacks. The goal here is to use function fingerprints of varying lengths to identify the location of a system call. This mechanism is used in more sophisticated attacks such as the slapper worm.

Our attack experiments used two distinct scanning attacks, one using a 12-byte signature and the other a 32-byte byte signature to identify the `execve` system call. The 12-byte signature was based on the identifying the two-byte sequence `0xcd80` (the binary encoding of the system call interrupt instruction '`int 0x80`') with the appropriate system call number, while the 32-byte signature used a byte sequence from the start of the function. The signature identification techniques, based on both substring and subsequence matching, used in our experiments was more sophisticated than techniques used in real exploits. The attack succeeded in the unprotected system and resulted in giving shell access. On the protected system the attack code was unable to locate the execve system call.

Hijacking a known system call
The final exploits were based on hijacking parameters to a known system call, i.e., replacing the arguments to a legitimate system call with different values of the attacker's choice. Our experiment scanned the program looking for the `execve` system call, and replaced its arguments to try and invoke a shell. The exploit succeeded on the unprotected system and resulted in giving shell access. On the protected system the attack was not able to identify the `execve` call or its parameters to perform replacement.

To demonstrate the efficacy of our proposed value and address signing techniques, we repeated the same exploit on an un-randomized binary. The location of the `execve` call was the same as in the original program so was the parameter ordering. Address and value signing were enabled and indicated in the signature. While the attack was successfully able to replace the parameters at user level and pass it in to the system call, the attack was detected by the kernel since the signatures did not match and the process terminated.

4.2 Effect on performance

The following set of experiments discuss the effect of signatures and randomization on the performance of a system based on the in-kernel interception approach. We begin with a description of micro-benchmarks which show that signatures impose a reasonable overhead. Next the effect on overall system performance is reported.

Micro-benchmarks
System call signing and fingerprint hiding via code randomization introduce two sources of overheads: the decryption and verification costs, and potential overheads due to increased argument passing, since each system call now takes six arguments.

Table 2 presents the overheads introduced by these techniques on a per system call basis. To measure the effect of these techniques on individual system calls each system call was executed in a tight loop of 10,000 iterations, and the total number of cycles taken measured using the Pentium processor's `rdtsc` instruction, which reads a 64-bit hardware cycle counter. The last row of Table 2 indicates the measurement overhead – the difference between two consecutive rdtsc instructions. Each experiment was repeated 12 times, the highest and lowest readings discarded, and the average of the remaining 10 readings are presented in Table 2. Column 2 gives the number of cycles required to execute an unmodified system call on an unmodified kernel; columns 3 and 4 show the effect of code randomization alone; columns 5 and 6 show the effect of system call signing and fingerprint hiding; and columns 7 and 8 show the effects of adding a decryption cache as discussed in Section 3.3.

It can be seen, from Table 2, that the effect of code randomization alone is small, typically ranging from 1.1% to 6.9%. The largest percentage increases are obtained for `brk` and `getpid`, and the smallest for `read`. Not surprisingly, when signatures are added, the cost increases noticeably, ranging from 9.1% for `read()` to about 50% for `brk()`. However, much of this increase can be recovered by adding a decryption cache, which amortizes the in-kernel decryption cost over several system calls, and thus brings the costs down to essentially that for code randomization alone.

Our experiences with a prototype system based on the interception in the Xen virtual machine are similar. Though, in this case disabling the "fast trap" mechanism led to higher overheads in the signed versus unsigned system call cost : as expected the highest percentage increase was for `getpid` (245%) and the smallest for `read` (16%).

Effect on overall application runtime
To discuss the effect of these techniques on overall performance of applications, we compare the running times of the original program and the protected version. 14 applications as described in Table 3 were selected for creating a benchmark suite. These programs are classified as either CPU or system call intensive as shown in the table: the CPU-intensive programs are from the SPECint-2000 benchmark suite, while the system call intensive programs are a collection of common applications that incur a large number of system calls. The programs were compiled using *gcc 3.2.2* at optimization level `-O0`, with additional flags to create statically linked relocatables which were then processed using our binary rewriting tool, PLTO. Three types of executables were created using PLTO: *untransformed binaries* corresponding to the unmodified program, *randomized binaries* in which the dead code insertion and layout transformations were performed and finally *signed binaries* in which apart from randomization, system calls were signed using DES encryption.

Our experiment consisted of measuring the time taken for each program to execute a fixed set of inputs. The `time` utility was used to measure the amount of time taken by each program and the amount of time per program computed as a sum of the user time and system time. Each program was repeated 12 times, the highest and lowest measurements discarded and the mean of the

System Call	*Original* Cost (cycles)	*Randomization* Cost (cycles)	*Randomization* Overhead (%)	*Rand. + Signatures* Cost (cycles)	*Rand. + Signatures* Overhead (%)	*Signature caching* Cost (cycles)	*Signature caching* Overhead (%)
getpid()	1263	1328	5.1	1880	48.8	1332	5.5
gettimeofday()	1470	1572	6.9	2159	46.9	1575	7.1
read(4096)	5841	5874	5.6	6370	9.1	5828	0.0
write(4096)	23946	24210	1.1	29876	24.8	25943	8.3
select()	2083	2176	4.5	2754	32.2	2267	8.8
brk()	1247	1333	6.9	1876	50.4	1325	6.2
rdtsc cost	84	88		88		88	

Table 2: Micro-benchmarks : Effect of transformations on individual system call performance

Program Name	Type	Description
bzip2	CPU	file compression program from SPEC INT 2000 benchmark.
gzip-spec	CPU	file compression program from SPEC INT 2000 benchmark.
crafty	CPU	Game playing (Chess) program from SPEC INT 2000 benchmark
mcf	CPU	combinatorial optimization program from SPEC INT 2000
vpr	CPU	FPGA circuit and routing placement from SPEC INT 2000
twolf	CPU	Place and route simulator from SPEC INT 2000
mpeg_play	syscall	Video decoder and player
pftest	syscall	Tester for the physical layer of a database
webserver	syscall	Simple single threaded webserver
copy	syscall	File copying program
gcc	syscall & CPU	Gnu C compiler from SPEC INT 2000
vortex	syscall & CPU	Object oriented database from SPEC INT 2000
pyramid	syscall	Multidimensional database - index creation and range queries
gzip	syscall	file compression program

Table 3: Benchmark suite

remaining 10 experiments computed. This value is reported in Table 4.

As seen in Table 4 randomization does not impose a significant performance overhead: on an average the performance overhead for the 14 applications was about 4.07%. Performing system call signing on randomized binaries increases the average overhead to 5.44%. Note that in some of the CPU intensive benchmarks the average time taken for the execution of signed and randomized binaries was less that of the unsigned binaries. This can be explained by the fact that the installer randomizes each program in a different way. `pftest` was the only program where either of the techniques imposed an overhead of greater than 10 %. This unusual overhead can attributed to the relatively small execution time of the `pftest` program. The last column describes the effect of adding a decryption cache. Not surprisingly the benefits of caching are reflected more in the system call intensive programs. `pyramid` shows the greatest impact and the overhead is reduced to less than half of the original value.

Effect on code size

Another side-effect of code randomization is an increase in the program code size. Table 5 describes the percentage increase in program code size computed by using the `objdump -d program | wc -c` command. The numbers presented in columns 2 and 3 of Table 5 indicate the number of number of bytes in the text sections of the binary.

On an average program size increased by about 4.7%, with gcc showing the largest increase (9.75%) and vortex the least (1.58%). Even though the frequency of junk code insertion was relatively high, at least 1 junk instruction per 4 instructions, the size of the junk instructions (typically 1 byte) was less than the average instruction length (about 2.9 bytes). This explains the smaller than expected increase in program code size.

Program	*Original*	*Randomization*		*Rand. + Signatures*		*Signature Caching*	
	Run time (secs)	*Run time* (secs)	*Overhead* (%)	*Run time* (secs)	*Overhead* (%)	*Run time* (secs)	*Overhead* (%)
bzip2	191.70	200.51	4.59	197.06	2.80	197.69	3.12
gzip-spec	152.34	155.48	2.06	154.45	1.38	153.88	1.01
crafty	107.57	114.03	6.00	114.31	6.26	114.25	6.20
mcf	236.38	242.09	2.41	242.29	2.50	239.41	1.28
vpr	220.44	226.45	2.72	235.81	6.97	229.32	4.03
twolf	396.94	408.89	3.01	408.71	2.96	407.41	2.64
mpeg_play	93.93	95.66	1.84	96.18	2.39	95.43	1.60
pftest	0.070	0.080	14.28	0.081	15.71	0.080	14.28
webserver	0.059	0.061	3.39	0.062	5.08	0.061	3.39
copy	0.88	0.92	4.54	0.94	6.81	0.92	4.54
gcc	90.21	95.43	5.78	97.61	8.20	94.58	4.84
vortex	165.55	171.86	3.81	173.43	4.76	172.65	4.29
pyramid	2.52	2.55	1.11	2.74	8.73	2.61	3.50
gzip	2.69	2.72	1.14	2.73	1.64	2.72	1.14
Average			4.07		5.44		3.99

Table 4: Performance overhead

Our final experiment measured the cost of performing address and value signing in the kernel. A copy program was set up by manually signing the parameters of the read and write system calls. Both the calls take in three arguments, the first an integer file descriptor, a char array and an integer representing the size. The values of the first and the third parameter along with the address of the second were used in computing the checksum. In the kernel the checksum computed is checked against these values. The average cost of each system call, measured by instrumenting the original program with `rdtscl` counters went up by about 200 cycles.

4.3 Discussion

This section briefly discusses various practical aspects with our approach. The first part of this section argues the pragmatism of this approach. Finally we present the attackers perspective and show that attacking any system with the intent of propagating the attack eventually comes to a dead end without the ability to perform system calls.

Usability and compatibility with tools
Our approach does not compromise the usability, maintainability or seriously hurt performance of existing applications. Debugging and development tools such as gdb, strace etc can continue to be used in the modified system. Modifying strace involved a handful of modifications in the strace source code. Our installer can easily be extended to support automatic upgrades and patches. The simplicity of the approach and its deployment ensure that unseen vulnerabilities, which may eventually be exploited to attack the system, are unlikely to be introduced.

Customizability
Our approach can be configured in several ways to adapt to deployment scenarios. Our technique is general enough to be uniformly applicable to a wide range of systems ranging from embedded devices to web servers. During deployment customizing the system is easy and a matter of choosing between different policies. For example, while our current implementation chooses to log a message and terminate processes that make unauthorized system calls, it is straightforward to change this, e.g., to restart the process, without changing the overall security model. The underlying algorithm can be also customized depending on the deployment site requirements, e.g., for cheap decryption, certificate verification can be chosen as an alternative to DES in an embedded context without affecting the protection offered by the approach.

Attacking the system
Preventing the attacker from issuing system calls severely restricts the capabilities of the attacker. The operations that can be performed, and in effect the damage possible is strictly restricted to within the address space of a given host process. In order to cross address spaces the attacker needs invoke a system call. Operations that can be performed by an attacker without issuing system calls, are restricted to scanning the processes address space and requesting intra-address-space functions. The OS maintains each process in its own address

Program	*Program Size* (bytes)			*Increase*
	Original	Randomization	Rand. + Signing	(%)
bzip2	5,667,562	5,846,338	5,857,143	3.34
gzip-spec	5,900,779	6,009,818	6,021,233	2.04
crafty	7,343,326	8,010,146	8,022,412	9.25
mcf	5,190,011	5,297,951	5,308,146	2.28
vpr	6,653,956	7,064,919	7,075,114	6.33
twolf	7,546,058	8,087,422	8,098,024	7.32
mpeg_play	15,802,993	16,715,412	16,736,194	5.90
pftest	5,271,729	5,396,150	5,408,077	2.59
webserver	5,899,141	6,064,041	6,077,953	3.03
copy	5,059,821	5,161,010	5,161,756	2.01
gzip	5,997,845	6,260,194	6,273,141	5.49
gcc	20,983,110	23,016,807	23,029,652	9.75
vortex	11,350,087	11,518,789	11,359,811	1.58
pyramid	5,264,248	5,466,860	5,475,263	4.01
Average				4.69

Table 5: Size overhead

space and not only regulates any communication across address spaces, but also regulates all changes to system state. Performing any operation such as installing a backdoor, giving shell access etc changes the state of the system and hence requires OS interaction. Since the attacker is not able to issue system calls he will not be able to change the state of the system and all his damage is restricted to the process that is targeted.

5 Related Work

A number of projects have proposed a variety of obfuscation techniques to prevent code injection attacks from making system calls. Chew and Song proposed a number of techniques, including permuting system call numbers, to make the system call interface less vulnerable to attack code [14]. Our approach goes well beyond such techniques, for example, instead of just obfuscating the system call numbers, our signing approach makes it possible to protect not only the system call number, but also additional attributes of the system call, such as its location, the values of known arguments, etc. Note that this makes it possible to handle attacks that exploit existing system calls in a program by changing their arguments, e.g., by sending the name of a different file argument to *open*. Finally, it allows the OS kernel to detect and deal intelligently with signature mismatches, e.g., by terminating the responsible process or signaling an intrusion; by contrast, with a simple permutation of system call numbers, a system call invoked from the attack code has unpredictable (and, potentially, problematic) results. Bhatkar *et al.* propose the use of address obfuscation to foil known-address attacks [10]. 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*. [43] is a similar technique based on dynamically and randomly relocating contents of a program's address space during startup. While these techniques are effective against known address attacks, they are susceptible to the scanning attacks described in this paper. We view these as complementary to ours and are investigating their application in extending our approach to handle local and insider attacks.

There is a wide body of literature on defending against code injection attacks. A number of researchers have proposed static program analysis to detect potential vulnerabilities such as buffer overflows [20, 26, 38]. When applied thoroughly, such schemes have the advantage of not letting an attacker even to 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 [17] and FormatGuard [16], 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., [11, 31]). There has been some recent work on disrupting the actual execution of attack code by means of "instruction set randomization"

[8, 24], but current proposals for this have the drawback of high execution overheads in the absence of specialized hardware support. Rabek *et al.*, propose monitoring the origin of library calls for the Windows operating system to prevent misuses of critical functions [30]. 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.

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.* [18, 23, 40] and subsequently explored by a number of researchers (see, for example, [25, 34, 37]). A drawback to this approach is that it is vulnerable to specific mimicry attacks [39]. 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 [9]. 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. Furthermore, all of the above approaches require the kernel to maintain fair amount of state information related to each application, while our approach only requires the kernel to maintain one key.

The use of NOP-insertion and code layout randomization to obfuscate code structure were proposed by Forrest *et al.* [19]; however, this work does not describe an implementation or provide experimental results. Other work along these lines is that of Wroblewski [42]. Many of these ideas can be traced to Cohen's work on system diversification [15]. Additional techniques for binary obfuscation, to hamper static disassembly, are described by Linn and Debray [28].

Imposing application specific restrictions through sandboxing is another technique that can be considered as related work. System call signing can be viewed as a simple yet efficient way of implementing a sandbox. In comparison to existing techniques [21, 22], our implementation offers significant speedups. Performance Signatures as have been proposed for intrusion detection can be viewed as extensions of address and value signing. While performance signatures rely on monitoring dynamic behavior of a program, our techniques rely inferring values through static program analysis. Our techniques can also be viewed as orthogonal yet relevant to the containment approaches [41, 29] being proposed to handle propagatory worm attacks. While these approaches emphasize regulating the network interfaces, our focus has been on the host's system interface. Most containment techniques try to throttle propagation from an already infected host; by contrast, our goals are to make it difficult to infect a host.

6 Conclusions

The increasing prevalence of remote code injection attacks against computer systems makes it increasingly important to develop effective countermeasures against such attacks. A crucial aspect of such attacks is that, in order to do any real damage, the attack code must invoke one or more system calls: if the attack code can be prevented from successfully invoking system calls, the attack will fails. This paper describes a novel defense against code injection attacks that uses a combination of cryptographic and compiler-based methods to harden the system call interface. The idea is to use cryptographic signatures to protect the arguments to system calls, together with compiler-based code randomization techniques to hide code fingerprints that could be used for mimicry attacks. We have evaluated our ideas using two experimental systems: one, suitable where source code for the OS kernel is available, where the operating system kernel is modified to decrypt and validate system call signatures; and another, suitable where source code is not available, where this is carried out using a virtual machine monitor. Our experiments indicate that our approach is effective in thwarting code injection attacks at modest cost.

References

[1] Apache worm source code. http://dammit.lt-/apache-worm/.

[2] Fuzz unix virus - source code. http://www.safenetworks.com/BSD/virus2.html.

[3] Open source vulnerability database. http://www.-osvdb.org/.

[4] Packet storm security. http://www.packetstormsecurity.nl/.

[5] Security focus bugtraq mailing list. http://www.-securityfocus.com/.

[6] Stackshield. http://www.angelfire.com/sk/stackshield.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield. Xen and the art of virtualization. In

Proc. ACM Symposium on Operating Systems Principles (SOSP), pages 164–177, October 2003.

[8] 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 *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 281–289, 2003.

[9] 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.

[10] 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.

[11] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[12] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. Eighth Workshop on Hot Topics in Operating Systems*, pages 125–132, 2001.

[13] CERT Coordination Center. Overview of attack trends. http://www.cert.org/archive/pdf/attack_trends.pdf.

[14] 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, December 2002.

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

[16] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proc. 10th Usenix Security Symposium*, August 2001.

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th. USENIX Security Symposium*, pages 63–78, January 1998.

[18] 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.

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

[20] 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.

[21] 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.

[22] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, CA, USA, 1996.

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

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

[25] 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.

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

[27] K. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th. USENIX Security Symposium*, pages 81–92, August 2002.

[28] C. Linn and S.K. 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, October 2003.

[29] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM*, 2003.

[30] J. C. Rabek, R.I. 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.

[31] G. Richarte. Bypassing the stackshield and stackguard protection: Four different tricks to bypass stackshield and stackguard protection. Technical report, Core Security Technologies, April 2000. http://www2.corest.com/corelabs/-papers/index.php.

[32] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, Feb 1978.

[33] 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.

[34] 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.

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

[36] Scan of the month number 25 The Honeynet project. Slapper-b worm. http://www.-honeynet.org/scans/scan25/.

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

[38] 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, February 2000.

[39] 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.

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

[41] M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *HPL-2002-172 Technical Report*, 2002.

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

[43] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.

[44] 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.