Authenticated System Calls

Mohan Rajagopalan
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
Email: *mohan@cs.arizona.edu*

Matti Hiltunen Trevor Jim Richard Schlichting
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932
Email: {*hiltunen,trevor,rick*}*@research.att.com*

Abstract

System call monitoring is a technique for detecting and controlling compromised applications by checking at runtime that each system call conforms to a policy that specifies the program's normal behavior. Here, a new approach to system call monitoring based on authenticated system calls is introduced. An authenticated system call is a system call augmented with extra arguments that specify the policy for that call and a cryptographic message authentication code (MAC) that guarantees the integrity of the policy and the system call arguments. This extra information is used by the kernel to verify the system call. The version of the application in which regular system calls have been replaced by authenticated calls is generated automatically by an installer program that reads the application binary, uses static analysis to generate policies, and then rewrites the binary with the authenticated calls. This paper presents the approach, describes a prototype implementation based on Linux and the PLTO *binary rewriting system, and gives experimental results suggesting that the approach is effective in protecting against compromised applications at modest cost.*

Keywords: Intrusion tolerance, operating systems, security policy, sandboxing, compiler techniques

1. Introduction

System call monitoring is a widely used technique for detecting compromised applications and for sandboxing applications to minimize the damage they can cause if they become compromised [2, 4, 5, 8, 9, 11, 14, 15, 18, 19, 20, 22, 24]. The intuition is that a compromised application can only cause real damage by exploiting system calls, making this interface the ideal point to detect and control attacks. The approach is based on having a model or policy of an application's normal system call behavior and then halting execution when an application deviates from this normal behavior. Policy checking and enforcement are security-critical, and hence, are performed entirely in the kernel or in the kernel in conjunction with a protected user-space policy daemon.

This paper introduces *authenticated system calls*, a new technique for monitoring and enforcing system call policies. An authenticated system call is a system call with additional arguments that specify a policy that the system call should satisfy, and a message authentication code (MAC) that guarantees the integrity of the policy and other arguments to the system call. The policy and MAC are part of the untrusted application, but the MAC is computed with a cryptographic key that is available only to the kernel. At each invocation of an authenticated system call, the kernel uses the key to recompute the MAC, and only allows the call to proceed if this matches the MAC passed in by the application. Since the application never has access to the key, it cannot successfully create a new authenticated system call or tamper with an existing authenticated system call. This approach of dividing the functionality between the application and the kernel is novel, and contrasts with other approaches that either rely on user-space policy daemons [4, 5, 8, 11, 18, 22], or require large-scale changes to the kernel [2, 14, 19, 20]. In comparison with our approach, the former can have unacceptably high execution costs unless frequently-used system calls are special-cased for enforcement in the kernel, while the latter results in a more complex kernel and the associated increase in execution overhead.

The second key element of our approach is the automatic transformation of the application to replace each system call with the equivalent authenticated call. This is done by a *trusted installer* program that reads the application binary, uses static analysis to determine the appropriate policy for each call, and then rewrites the binary with the authenticated calls. The use of static analysis has significant advantages over methods based on hand-written policies or policies obtained by training, i.e., recording the system call behavior of the application over a period of time. In particular, it is completely automatic, produces policies quickly, and it does not miss system calls invoked by rarely-used parts of the application. We demonstrate these advantages

empirically by comparing our policies with those published elsewhere for the well-known Systrace system call monitoring system [15].

The primary goals of this paper are first, to describe the details of authenticated system calls, and second, to give experimental results from a prototype implementation of the approach based on Linux and the PLTO binary rewriting system [17]. In addition, we describe extensions that can be used to make policies more expressive. These include support for state-dependent policies and *capability tracking*, where information on linkages between argument values in different system calls is included in the policy.

2. Basic Approach

Overview. As noted above, two steps are needed to protect systems using our basic approach: transforming programs to replace system calls with authenticated system calls, and runtime checking by the kernel to ensure that each system call matches its policy. The first step, *installation*, is illustrated in figure 1. The binary of a program is read by a trusted installer program, which first generates the policy that captures the allowed behavior for each system call using static analysis, and then rewrites the binary so that each system call includes the policy and a cryptographic MAC that protects the policy. The key for the MAC is specified during the installation process. The second step, *syscall checking*, is illustrated in figure 2. At runtime, each system call is intercepted by the kernel and, after verifying the MAC using the same key as used during installation, the behavior of the call is verified against the policy. If the behavior matches the policy, the call is allowed; otherwise, the call is rejected and the executing process terminated.

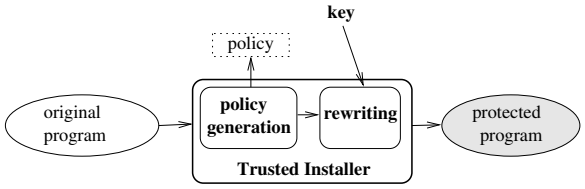

Figure 1. Program installation

Here, we elaborate on the details of policies, installation, and syscall checking for this basic approach. This description matches our prototype implementation, which is used for the experimental results in section 3. The most important limitations of this prototype are that it currently requires relocatable binaries and can handle only statically-linked executables; ways to address both of these issues are discussed further below and in section 4, respectively. Techniques for improving the expressiveness of the automatically-generated policies are also presented in section 4.

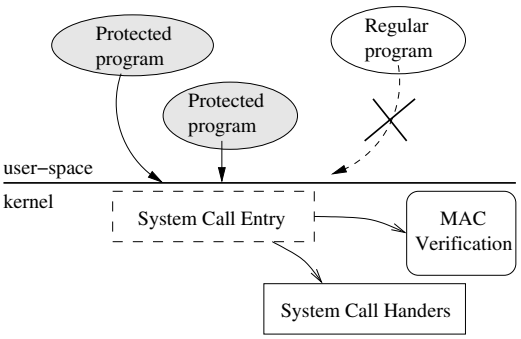

Figure 2. Syscall checking

Policies. A policy can be defined as the set of verifiable properties of a system call request. Our current prototype enforces system call policies of the following form:

```
Permit open from location 0x806c462
 Parameter 0 equals "/dev/console"
 Parameter 1 equals 5
```

This policy says that an application can invoke the open system call from the call site at memory address 0x806c462, provided that the first parameter is a pointer to the string "/dev/console," and the second parameter is the constant 5. In general, our policies can specify the system call number, the call site, constant parameter values (e.g., integer constants), and constant parameter addresses in the read-only text segment (e.g., strings). If a policy does not give a value for a parameter, then the parameter is unconstrained and any value is allowed.

These policies are reasonably expressive: most of the published system call policies for other system call monitoring systems such as Systrace constrain only the system call number, and constant parameter values and addresses. In section 4, we extend policies to include, for example, policies derived from call graphs, policies that allow argument values to match patterns, and capability tracking policies for arguments such as file handles.

Installation. The trusted installer program is used by a security administrator to generate the policy for an application, and to produce an executable binary that contains authenticated system calls. The installer reads in an application binary and disassembles it into an intermediate representation. Then, the installer determines system call arguments (using standard compiler techniques such as constant propagation [1]) resulting in a policy for each system call consisting of the system call number, call site, and some argument values. We call such a policy the system call's *authenticated system call (ASC) policy*, while the combination of ASC policies for all system calls in an application make

up the application's ASC policy. Once an application's policy has been generated in this way, it can be printed out for the administrator to review, or the installer can proceed directly to the next step, rewriting.

In the rewriting step, the installer transforms the binary by replacing the original system calls with authenticated system calls. An authenticated system call consists of the original system call extended by two arguments: a *policy descriptor* and the MAC. The policy descriptor is a single 32-bit integer value that describes what parts of the system call are protected by the MAC. In particular, for each original argument of the system call, it encodes whether the argument is unconstrained or constrained to be a constant value or address as defined above.

The installer computes the MAC over the *encoded policy*, i.e., a byte string that is a self-contained representation of the policy. It builds this encoded policy by concatenating the system call number, the address of the call site, the policy descriptor, and the argument values for those arguments that are constrained. For example, for a policy

```
Permit fcntl from location 0x806c57b
 Parameter 1 equals value 2
```

the installer computes the byte string

```
005c 00000011 0806c57b 0000002
```

Here 005c is the system call number of fcntl, 00000011 is the 32-bit number that says that the call site and parameter 1 should be constrained and parameter 0 should be unconstrained, 0806c5b7 is the call site, and 00000002 is the value for parameter 1. The installer computes a MAC over this byte string using a key provided by the security administrator. The prototype uses the AES-CBC-OMAC message authentication code, which produces a 128-bit code [10]. The installer adds the MAC to the data segment of the binary, and adds a pointer to the MAC as an argument to the system call. The result is an authenticated system call, with two more arguments than the original system call.

The installer completes once it has processed every system call in the program. The system as a whole is protected once all binaries that run in user space have been transformed to use authenticated system calls by the installer.

Syscall checking. Enforcement of an application's ASC policy is done by the kernel at runtime. When an authenticated system call occurs, the kernel receives arguments that include the system call number, the arguments to the original unmodified call, the policy descriptor, and the MAC. Furthermore, it can determine the call site based on the return address of the kernel interrupt handler. Using this information, the kernel performs the following computation to validate that the actual system call complies with the specified policy. It first constructs an encoding of the policy by concatenating the system call number, the call site, the policy descriptor, and those argument values that are specified in the policy descriptor. The kernel then computes a MAC over this encoding using the same key used during installation, and checks that the result matches the MAC passed in as an argument. If the MACs match, the kernel carries out the system call; otherwise, it terminates the process, logs the system call, and alerts the administrator. Unauthenticated calls are also blocked.

Syscall checking is designed so that MAC matching fails if an application has been compromised. Note that the arguments to the authenticated system call are under the control of the application, which means that it might have tampered with any of them, including the policy descriptor and MAC, or it might have even tried to construct a new authenticated system call somewhere in the heap. However, any change to the system call number, call site, policy descriptor, or values of arguments constrained by the policy would result in a change to the encoded policy that is constructed by the kernel. This in turn would change the MAC needed to pass the kernel test. Our cryptographic assumption is that it is infeasible for the adversary to construct a matching MAC for its changes without access to the key, hence, any attempt by the application to change the system call to violate the policy will fail.

Prototype implementation. Our prototype implementation of the trusted installer is based on the PLTO binary rewriting system that reads a binary executable, constructs an intermediate representation of the program and its control flow, performs optimization passes on the intermediate representation, and finally rewrites the binary for the optimized program [17]. The installer functionality is added to PLTO as optimization passes that determine system call policies and then replace each system call in the program with an authenticated system call. The installer runs on Linux, PLTO's native platform. The policy generation portion of the installer has also been ported to OpenBSD to compare policies generated on the two platforms; this is used for the experimental evaluation in the next section.

PLTO is an optimization tool, and, as a result, it requires relocatable binaries (i.e., binaries in which the locations of addresses are marked), so that addresses can be adjusted as code transformations move data and code locations. Our installer currently inherits this requirement, although it should be straightforward to generate policies for binaries without relocation information. One impact of this restriction is that the binaries we test in the next section had to be compiled from source, since binaries shipped with standard Linux and Unix distributions do not contain relocation information.

Syscall checking has been implemented in Linux by adding a little over 200 lines of code to the kernel's software trap handler, and including a cryptographic library of

about 3000 lines of code for MAC functionality [7]. The software trap handler is responsible for identifying the system call number and arguments, invoking the appropriate system call handler, and returning the result to the calling application. We modified the handler to call a routine that uses the MAC to verify that the system call satisfies the required policy. We have not yet implemented syscall checking in OpenBSD.

To evaluate the security provided by authenticated system calls against different forms of code injection attacks, we devised a set of synthetic attack experiments. Exploits based on techniques such as code injection and parameter hijacking were used against a synthetic program that contain a number of exploitable vulnerabilities, such as overflowable buffers. The attack code was able to cause the program to spawn a root shell on the unprotected system, while the attacks failed against the version using authenticated calls.

3. Experimental Evaluation

This section gives the results of an experimental evaluation of authenticated system calls as realized in our prototype implementation. We start by describing the ASC policies generated by the installer, and comparing them with some of those available for Systrace. We then present experimental results that quantify the performance impact of authenticated system calls compared with unprotected calls.

Policies. An ideal policy would permit the system call behaviors needed for normal operation and no others. If the policy permits system calls not used by the uncompromised application (*unneeded calls*), it leaves open the possibility that such calls could be exploited by an attacker. If the policy omits some system calls actually used by the application (*needed calls*), it raises the possibility of a false alarm that causes the application to be terminated unnecessarily. False alarms are a significant administrative headache and barrier to use.

We generate our ASC policies through a conservative static analysis, so our polices include all needed calls, and thus avoid false alarms. In contrast, policies generated by hand or by training may miss needed system calls, for example, because they occur only in parts of the program that are rarely executed.

ASC policies might include unneeded system calls, because no static analysis is able to identify the exact set of needed calls for every program. Note, however, that unneeded calls might also appear in policies produced by hand or by training. Hand produced policies can include mistakes, for example. System calls identified through training are never unneeded, by definition, but there are still opportunities for errors; for example, policies might be obtained by training on one version of an application and operating system, and used on another. In general, policies generated by training are not portable between operating systems, or even between different versions of the same operating system, and they may need to be adjusted even when only libraries are updated.

In order to gather some empirical evidence regarding false alarms, unneeded system calls, and operating system effects, we ported our policy generator from Linux to OpenBSD. OpenBSD is a useful test case because it supports a system call monitor, Systrace, in its default build, and researchers have published Systrace policies for OpenBSD applications. The Systrace policies are generated through training along with hand edits.

Program	ASC policy for Linux	ASC policy for OpenBSD	Systrace policy for OpenBSD
bison	31	31	24
calc	54	51	24
screen	67	63	55

Table 1. Number of system calls in policies

Table 1 compares the number of distinct system calls permitted in both ASC and Systrace policies for several common Unix programs: bison, the GNU Project parser generator; calc, an arbitrary-precision calculator program; and screen, a screen manager with terminal emulation. The first column gives the numbers for the ASC policy generated on Linux, the second column the ASC policy generated on OpenBSD, and the third column gives the numbers for Systrace policies published by the Project Hairy Eyeball web site [6]. This rough comparison illustrates two things:

- There are significant differences in the system calls needed for the same application running on different operating systems; this implies that policies for one operating system cannot simply be used on another.
- ASC policies identify system calls that are not present in Systrace policies.

Table 2 examines the policies for bison in more detail. The table shows system calls that are permitted by the ASC policy generated on OpenBSD but not by the Systrace policy, and vice versa. Note that the ASC policy includes many system calls that are not present in the Systrace policy. We believe that most of these calls are in fact needed, and we have verified some of them by hand using a system call tracer on actual runs of applications. This means that the Systrace policy can cause false alarms.

Conversely, there are a few system calls permitted by the Systrace policies that are not allowed in the ASC policy. They break down as follows.

System call	ASC	Systrace
__syscall	yes	NO
close	NO	yes
fcntl	yes	NO
fstatfs	yes	NO
getdirentries	yes	NO
getpid	yes	NO
gettimeofday	yes	NO
kill	yes	NO
madvise	yes	NO
mkdir	NO	yes (fswrite)
mmap	NO	yes
nanosleep	yes	NO
readlink	NO	yes (fsread)
rmdir	NO	yes (fswrite)
sendto	yes	NO
sigaction	yes	NO
socket	yes	NO
sysconf	yes	NO
uname	yes	NO
unlink	NO	yes (fswrite)
writev	yes	NO

Table 2. Comparison of policies for bison

mmap. The mmap system call is implemented on OpenBSD by invoking __syscall, a generic indirect system call function. The ASC policy correctly constrains the arguments of __syscall so that only mmap can be invoked, however. With Systrace, this indirection is hidden from users since its policy does not explicitly allow __syscall.

close. The call of close is not identified by PLTO due to an unusual implementation on OpenBSD that PLTO currently cannot disassemble. However, PLTO always reports when it cannot completely disassemble a binary, so that the administrator would always be aware of such a problem. To date, we have not encountered similar difficulties on Linux, PLTO's native platform.

mkdir, readlink, rmdir, unlink. The Systrace system uses two generic names, fsread and fswrite, to specify sets of system calls; fsread denotes read-related system calls and fswrite denotes write-related calls. The fact that mkdir, etc., are not in the ASC policy indicates that they are unneeded system calls, but their execution would be allowed with Systrace since its policy includes fsread and fswrite.

Next, we examine the degree to which each authenticated system call is protected from alteration by its MAC. In our current prototype, the system call site and call number are always protected by the MAC, as are those arguments whose values can be determined by static analysis. It is, of course, impossible to determine all argument values using such techniques; for example, the value may be read as a user input, generated as a result of a system call, or may be unknown because of pointer aliasing. In practice, however, static analysis can determine enough values to be useful [22], and it can provide a partial policy template that can then be extended by the security administrator using dynamic profiling and application knowledge.

prog	sites	calls	args	o/p	auth	mv	fds
bison	158	31	321	31	90	2	69
calc	275	54	544	78	183	2	109
screen	639	67	1164	133	363	7	297
tar	381	58	750	105	238	3	152

Table 3. Argument coverage

Table 3 provides the results of generating ASC policies for four programs: the three from above and tar, the Unix archiving program. The *sites* column indicates the number of separate system call locations in the program, *calls* indicates the number of different system calls, and *args* gives the total number of arguments (not including the system call number) from all the call sites. The *o/p* column gives the number of system call arguments that are output-only arguments, that is, the argument is an address of a structure where the kernel stores the result of the call. The *auth* column lists the number of arguments that could be determined by the static analysis done by the installer and that could be authenticated by the basic approach. These results indicated that 30–40% of the arguments can be protected based on static analysis and the basic approach.

In addition to these arguments, there are many others that might be protected by using extensions such as those described in section 4. Table 3 includes statistics for two of these as well: arguments where the value can be determined using static analysis but each argument may have two or more values (*mv*), and arguments that are file descriptors that were returned previously from system calls such as open or socket (*fds*).

Performance. This section measures the performance overhead introduced by the syscall checking mechanism. We begin with a description of results from microbenchmarks that measure the impact on individual system calls, and then measure the effect on the overall execution time for a number of programs.

Table 4 presents the overheads introduced by these techniques on a per-system-call basis. These results were obtained by executing each system call 10,000 times using a loop, and measuring the total number of CPU cycles using the Pentium processor's rdtsc instruction, which reads a 64-bit hardware cycle counter. The last two rows indi-

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
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
gzip	syscall	file compression program

Table 5. Benchmark suite

System Call	*Original* Cost (cycles)	*Authenticated* Cost (cycles)	*Authenticated* Overhead (%)
getpid()	1141	5045	342.2
gettimeofday()	1395	5703	308.8
read(4096)	7324	10013	36.7
write(4096)	39479	40396	2.3
brk()	1155	5083	340.1
rdtsc cost	84	84	
loop cost	4	4	

Table 4. Effect of authentication

Program	*Original* *Time* (secs)	*Original* *Std.* Dev.	*Authenticated* *Time* (secs)	*Authenticated* *Std.* Dev.	*Authenticated* *Overhead* (%)
bzip2	196.80	1.46	198.56	2.67	0.89
gzip-spec	155.38	0.14	156.39	0.19	0.65
crafty	108.32	0.15	108.39	0.27	0.06
mcf	240.96	8.22	244.96	1.35	1.66
vpr	221.25	1.24	228.25	3.38	3.16
twolf	389.97	5.58	402.59	8.38	3.24
gcc	92.88	1.19	93.97	0.74	1.17
vortex	3.80	0.01	3.91	0.01	2.89
pyramid	0.99	0.01	1.02	0.01	3.03
gzip	2.78	0.03	2.82	0.03	1.01
Average					1.78

Table 6. Performance overhead

cate the overhead of the measurement process itself. Each experiment was repeated 12 times; the highest and lowest readings were discarded, and the average of the remaining 10 readings is used in the table. Column 2 gives the number of cycles required to execute an unmodified system call on an unmodified kernel, while columns 3 and 4 show the effect of authenticated system calls.

The results indicate a noticeable cost for the checking mechanism, about 4000 cycles for each call. As might be expected, however, on a percentage basis, the overhead is much more significant for simple system calls such as getpid and gettimeofday than for more complex calls like write, where the costs associated with buffering and memory accesses dominate.

To measure the effect of these techniques on the overall performance of applications, we compared the running times of 10 programs and their protected counterparts (table 5). These programs can be 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 make a large number of system calls. The programs were compiled using gcc 3.2.2 into statically linked relocatables that were then processed using our binary rewriting system, PLTO. Two types of executables were created: *unauthenticated binaries* corresponding to the unmodified program and *authenticated binaries* that use authenticated system calls. We use unauthenticated binaries generated by PLTO rather than simply gcc as the baseline, since PLTO itself applies certain optimizations such as dead code elimination, basic block layout, and instruction scheduling. As a result, applying these optimizations in both cases gives the most accurate representation of the actual cost of an authenticated call. The cost of transforming the programs including PLTO optimizations ranged from 3.19 seconds for mcf to 85.37 seconds for gcc.

Our experiment consisted of measuring the time taken for each program to execute on a fixed set of inputs. The `time` utility was used to measure the time taken by each program, with the total computed as the sum of the user and system time. As before, each experiment was repeated 12 times; the highest and lowest readings were discarded, and the average of the remaining 10 readings is used in the table. The results, reported in table 6, indicate a modest overhead ranging from 0.06% to 3.24%.

Our final experiment studies the effect of the authenti-

cation mechanism on a multi-program benchmark. This benchmark is similar to the Andrew Benchmark and consists of a series of tasks that perform routine operations such as file creation, directory creation, file compression, file archival, permission checking, moving files, deleting files, and sorting the content of files. Each iteration of the benchmark results in the invocation of about 12000 system calls. Authenticated versions of several general purpose tools such as gzip, gunzip, rm, chdir, mv, chmod, tar, cat, and cp were used to perform the tasks. The execution time of the benchmark using original binaries was 258.68 seconds, while the execution time for authenticated binaries was 261.50 seconds, an increase of only 1.09%.

It is difficult to compare the overhead of authenticated system calls with other system call monitors because each system enforces different policies. Note, however, that the total overhead of our approach is well below that of other systems, even though we do policy checking on *all* system calls, unlike, for example, Systrace [15] and Ostia [5].

4. Improving Policies

This section describes techniques for making policies more expressive to allow, for example, more complete argument coverage. We have not yet implemented these techniques, but we anticipate that they will all be relatively straightforward extensions to the existing system.

Meta-policies and policy templates. An *ASC meta-policy* is a specification that dictates how strict a policy is required for each system call. In particular, for each system call, the meta-policy indicates whether the call site must be specified in the policy and which arguments of the system call must be constrained. Compared with our basic approach, meta-policies focus on what *must be* protected for a system call rather than what *can be* protected automatically based on static analysis. Meta-policies would typically be derived from the threat level of different system calls [2] and local administrative policies.

The meta-policy is given as input to the installer along with the original program (figure 3). If the policy generator cannot determine all the argument values required by the meta-policy based on static analysis, it generates a *policy template* with spaces for the additional required arguments. An administrator can then hand-specify a value or a pattern (e.g., "/home/smith/www/*") for an argument based on application knowledge or dynamic profiling. The result of this is the *complete ASC policy*, which is used during the rewriting phase by the installer.

Patterns in meta-policies are implemented by having the installer store the patterns in the program address space. Patterns are protected and handled like constant value strings: patterns are stored in the read-only text segment

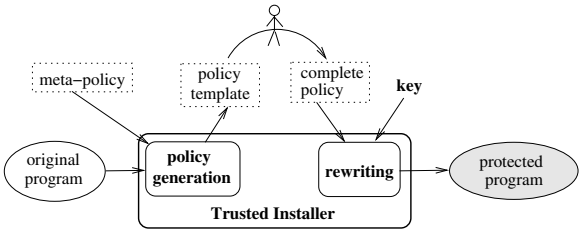

Figure 3. Meta-policies and policy templates

and the address of the pattern is included in the policy descriptor that is protected by the MAC. The kernel checks the MAC to verify that the policy and the patterns have not been modified, and uses standard pattern matching routines to match the argument runtime value against the pattern. Or, program checking techniques might be used to do the pattern matching in the untrusted application, with a quick verification by the kernel.

Meta-policies also play a role in extending authenticated calls to handle dynamic libraries. With dynamic libraries, system call sites for calls within the library are not known until the library is loaded at runtime. This means that our basic approach cannot protect the call site from alteration using a MAC, as is done with statically-linked binaries. In addition, arguments used by system calls in dynamic libraries are often passed as arguments to library functions, meaning that their values cannot be determined by static analysis.

Dynamic libraries are processed based on the security requirements stated in the meta-policy as follows. The dynamic libraries on a machine are installed first before the application programs. During this process, if a system call in a dynamic library function cannot satisfy the meta-policy—that is, static analysis cannot generate a complete policy—the specific function is removed from the dynamic library and set aside for static linking with application programs that require the function. Once this has been done for all system calls in the library, the functions that remain have their system calls transformed into authenticated calls in the same manner as before. Functions in this new protected dynamic library can then be loaded at runtime. Note that since a single meta-policy is used for the installation of each dynamic library, it must be something that is appropriate for all applications that use that library on a given machine.

Policies with state. Another useful feature is to allow policies that rely on state of some sort. For example, one might want a policy that requires that each call to open must be followed by a close before open can be called again. Here, the state would be a boolean indicating whether open is allowed. The state variable would be checked and modified by the syscall checker when an open is called, and mod-

ified again when close is called.

An obvious way to support policy state is to store the state in the kernel. However, one of the virtues of authenticated system calls is that they require minimal change to the kernel, something that would be lost if the state is large or has a complex structure. Therefore, we would like a way to keep any policy state in the application itself, with only the updates and maintenance being done by the kernel.

This can be achieved using the idea of *on-line memory checkers*, where a data structure is stored in unreliable memory, and a trusted checker with a small amount of reliable memory verifies the correctness of each update as it occurs [3]. Assume that some per-process state in the form of a byte string is required to implement policy state. Then, we modify the basic authenticated system call approach as follows. First, the kernel is modified to maintain a single counter variable for each process, initialized to 0 and stored in kernel space. Then, the installer is changed to add one variable to the data segment of each application to hold the policy state (the byte string), and a second variable to hold a MAC for the state. The state variable is initialized as needed by the policy, and the state MAC is calculated over the initial state and the initial application counter value, 0. Pointers to the policy state and state MAC are then passed as additional arguments in each authenticated system call.

At syscall checking time, if the policy for the system call depends on the policy state, the kernel recomputes the state MAC using the application counter and the policy state passed in the call. If the recomputed MAC matches the state MAC passed in by the application, the call is allowed to proceed; otherwise, the application is terminated. If the policy requires changing the policy state, the kernel increments the application's state counter and calculates a new state MAC over the new counter value and policy state. The new state MAC is stored over the previous state MAC in application space.

Once again, our cryptographic assumption is that it is computationally infeasible for an adversary to compute a valid MAC for some desired policy state and state counter. The kernel-space state counter prevents the adversary from re-using state MACs computed by the kernel for previous states.

Call graph. A simple but useful example of a policy requiring state is one based on the application's call graph. Such a policy could, for example, require that the application's system call trace be a path in the call graph, providing further protection against compromised applications. Policies of this type are used by Wagner and Dean [22], who use static analysis to construct a conservative approximation of the call graph, which is then encoded as a finite automaton for syscall checking.

Policies of this type are easily implemented with authenticated system calls. The installer already computes the call graph of the system calls of an application. Given this call graph, we can label each node of the call graph by its call site. The policy state becomes the call site of the last node executed by the application. The policy of each system call is then extended to say that the policy state must be one of the predecessors of the system call in the static call graph. Syscall checking in the kernel is extended to verify that the previous call site is in the list of predecessors given in the policy, and to update the policy state to the new call site.

As was the case with the general issue of state-based policies, some of the work can be moved from the kernel to the application to minimize the impact on the kernel. For example, we could force the application to calculate the predecessor of the node from the list of possibilities, and pass this in to the kernel to verify.

Capability tracking. Another useful feature for policies is the ability to specify that an argument to a system call be based on arguments or return values of previous system calls. An example would be a policy for a read system call that requires that the file descriptor argument be a value returned by a previous open system call [20]. We call policies of this sort *capability tracking policies*, since such arguments are being used in a manner analogous to capabilities. We illustrate how the basic authenticated system call approach can be extended to support this feature using the example of file descriptor tracking.

A naive implementation of file descriptor tracking would use policy state to store the last file descriptor returned by each call to open. The policy for each read system call would specify that the file descriptor should match the file descriptor for the desired open system call. However, this ignores the fact that an open system call can be executed more than once, that more than one file descriptor returned by the open can be active at once, and that file descriptors can be reused after they have been closed.

A better approach is to store, for each open system call, a set of currently active file descriptors. The policy for each open then adds a file descriptor to the set, while the policy for close removes a file descriptor. This involves fairly complicated data structures, so we would not use the simple policy state implementation described above, but rather a more efficient implementation based, for example, on authenticated dictionaries.

5. Discussion

File name normalization. A recurring problem for system call monitors has been dealing with race conditions caused by features such as symbolic links and relative file names. For example, consider a policy that allows an application to open a temporary file, /tmp/foo. An attacker

could try to exploit this by creating a symbolic link named /tmp/foo that points to /etc/passwd, and then overwriting the password file by opening and writing /tmp/foo.

To avoid this, system call monitors often use the convention that a file name in a policy must refer to the *normalized* file name, that is, the name of the file after all symbolic links have been followed. While doing normalization correctly can be complex, strategies developed elsewhere for performing this step in the kernel during syscall checking [4] apply to our approach. In addition, we anticipate that it is possible to move some of the processing into the untrusted application, using techniques similar to those described above in section 4 for state-dependent policies.

Frankenstein attacks. An application protected by our approach can become compromised, for example, through a buffer overflow, giving an attacker control of the application process. The process would not be able to execute arbitrary system calls, but it could execute any authenticated system calls in the application, provided it did not change the call site and parameters covered by the policy. This can lead to mimicry attacks [23], which are well known and which can be defended against by using more precise policies.

Our current prototype implementation is vulnerable to a similar, but more subtle attack: the compromised application could execute authenticated system calls that it finds in *other* applications on the system. Once the attacker has control of an application, it might use it to examine the other applications on the system, and construct and execute a new application composed of authenticated system calls from many applications. We call this a *Frankenstein attack*.

Call graph policies can defend against such Frankenstein attacks. Recall that a call graph policy requires an application to execute system calls in an order consistent with its static call graph. The call graph of an application is self-contained, so if we impose a call graph policy on all of the applications, a Frankenstein program would be forced into executing only the system calls of a single application, namely, the application that supplies the first authenticated system call executed by the Frankenstein program. We only need to take care that the installer use distinct labels for the nodes of all the application programs.

Another kind of Frankenstein attack targets the string literals that can appear in policies. In the current implementation, a string literal that appears in a policy is encoded as its address. The MAC produced is therefore dependent on the address, and not the contents, of the string. We are relying on the memory protections of the operating system to prevent a rogue process from modifying its string literals. However, it might be possible for a rogue process to build an altered copy of itself, identical except for the contents of some of the string literals, and transfer control (via exec) to the copy, while respecting even a call graph policy. One way to prevent this would be to protect the string contents by a MAC rather than, or in addition to, the address. This must be done with care, however, as the adversary gets to choose the actual arguments to the system call, and could pass in a very long string or an inaccessible address in an attempt to disrupt the kernel system call checking code.

Related work. System call monitoring falls into the broader area of intrusion detection systems. An intrusion detection system can try to detect misuse (known attacks) or anomalies (deviation from normal behavior). Misuse detectors can be vulnerable to previously unknown attacks, while anomaly detectors can suffer from false alarms. Our system is an anomaly detector that avoids false alarms because of our conservative static analysis. The basic 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 originally proposed by Forrest *et al.* [9, 24].

System call monitoring can be implemented entirely in user space [12, 13], but typically this is not secure against attacks such as buffer overflows, so this is not appropriate for our setting. User-space implementations can be secure for applications written in a safe language such as Java [21]. However, most systems have focused on applications written in unsafe languages, so they are implemented entirely in the kernel [2, 14, 19, 20] or by using kernel hooks or patches in combination with a user-space policy daemon or monitor [4, 5, 8, 11, 18, 22].

Our implementation uses a kernel modification in combination with binary modifications to the untrusted user application itself, and does not rely on a separate policy daemon. Instead, we use cryptographic and program checking techniques to ensure that any work done by the untrusted application regarding policy decisions is done correctly.

In comparison to systems implemented entirely in-kernel, our kernel modifications are minor—a couple of hundred lines of code compared to thousands with other systems. A completely in-kernel implementation must maintain the policies and the logic for determining which policy applies to a given call; we place these burdens on the application. Note in particular that the exact policy for a given authenticated system call is provided by the call itself. This gives us an advantage in speed and simplicity.

In comparison to systems implemented with user-space policy daemons, we have the advantage of fewer context switches, leading to a very modest overhead. Avoiding a separate monitor process simplifies policy checking, because the operating environment (current working directory, etc.) does not have to be mirrored, some race conditions are avoided, and we do not have to protect against the user application killing the monitor.

Policies for most system call monitors are developed by

hand or by training; Wagner and Dean [22] is the only other system we are aware of that uses static analysis. Wagner *et al.* [22, 23] introduced mimicry attacks and suggested making policies more precise to combat them; authenticating system call arguments and using call graph policies are two of their suggestions that we use.

6. Conclusions

Attacks that attempt to compromise a computer system using the system call interface are an increasingly important threat. Monitoring system calls and disallowing those that do not conform to a program's security policy is an effective mechanisms for stopping a large class of such attacks. Essentially, a system call monitor can convert a potentially successful attack into a fail-stop failure [16] of the compromised process.

In this paper, we presented authenticated system calls, a novel approach to system call monitoring. This approach has been implemented using only small modifications to the kernel, without the need for heavyweight kernel data structures or the use of a user-space policy daemon at runtime. We also presented an automated approach for generating security policies based on static analysis, something that can be extended using other techniques if necessary.

We evaluated the approach on Linux and, for policy generation, on OpenBSD. In doing so, we provided measures of the effectiveness of policy generation and quantified the modest runtime impact of using authenticated system calls over unprotected ones. We also presented a number of extensions to the basic approach that can increase its effectiveness by improving the expressiveness of policies.

Acknowledgments

S. Debray provided valuable insights on the technical issues in this paper. This work was supported in part by NSF under grants EIA-0080123, CCR-0113633, and CNS-0410918.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] M. Bernaschi, E. Gabrielli, and L. Mancini. Operating system enhancements to prevent the misuse of system calls. In *ACM CCS*, pages 174–183, 2000.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *IEEE Symp. on Foundations of Computer Science*, pages 90–99, 1991.

[4] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Network and Distributed Systems Sec. Symp.*, 2003.

[5] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Network and Distributed Systems Sec. Symp.*, 2004.

[6] J. Geovedi, J. Nazario, N. Provos, and D. Song. Project hairy eyeball. http://blafasel.org/∼floh/he/.

[7] B. Gladman. AES combined encryption/authentication library. http://fp.gladman.plus.com/AES/index.htm.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Usenix Sec. Symp.*, 1996.

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

[10] T. Iwata and K. Kurosawa. OMAC: One-key CBC MAC, 2002.

[11] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *ISOC Network and Distributed Sec. Symp. (NSDD00)*, pages 19–34, 2000.

[12] M. Jones. Interposition agents: Transparently interposing user code at the system interface. In *ACM SOSP*, pages 80–93. 1993.

[13] E. Krell and B. Krishnamurthy. COLA: Customized overlaying. In *Winter USENIX Conf.*, pages 3–7. Jan. 1992.

[14] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *LNCS 2808*, pages 326–43, 2003.

[15] N. Provos. Improving host security with system call policies. In *USENIX Sec. Symp.*, 2003.

[16] R. Schlichting and F. Schneider. Fail-stop processors: An approach to designing fault tolerant computing systems. *ACM TOCS*, 1(3):222–238, Aug. 1983.

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

[18] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symp. on Sec. and Priv.*, pages 144–155, 2001.

[19] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *8th USENIX Sec. Symp.*, pages 63–78, 1999.

[20] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. *ACM SOSP*, Oct. 2003.

[21] V. Venkatakrishnan, R. Peri, and R. Sekar. Empowering mobile code using expressive security policies. In *Workshop on New Sec. Paradigms*, pages 61–68. 2002.

[22] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symp. on Sec. and Priv.*, pages 156–169, 2001.

[23] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detecion systems. In *ACM CCS*, pages 255–264, 2002.

[24] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symp. on Sec. and Priv.*, 1999.