A Practical Mimicry Attack Against Powerful System-Call Monitors

Chetan Parampalli, R. Sekar and Rob Johnson
Stony Brook University

Abstract

System-call monitoring has become the basis for many host-based intrusion detection as well as policy enforcement techniques. Previous work on mimicry attacks showed that system-call monitors can be evaded, but these attacks are effective primarily against relatively weak system-call monitors, e.g., those that ignore system-call arguments. As more powerful monitoring techniques continue to be discovered, a natural question is whether they too can be evaded. In this paper, we formulate a class of very powerful system call monitors, called *I/O-data-oblivious* monitors, which incorporate perfect knowledge about the values of all system call arguments as well as their relationships, with the exception of data buffer arguments to `read` and `write`. We then present a successful mimicry attack, which we call a *persistent interposition attack*, against such monitors. Hackers can build a persistent interposition attack from a typical code injection vulnerability with moderate effort and tools and techniques in use today — we have implemented working attacks against real-world applications such as the Apache web server. In contrast, previous work hasn't demonstrated working mimicry attacks on realistic applications protected by gray-box IDS.

1 Introduction

System-call monitoring intrusion detection systems (IDS[1]) have created an arms race between defenders, who create new and more powerful monitors, and attackers, who create new attacks to evade the monitors. Researchers have refined system-call monitoring defenses by capturing sequencing relationships [17, 36], by using call-site [30], call-stack [9, 11], and system-call argument [21, 32, 5] information. In parallel, *mimicry attacks* have been refined to evade many of these defenses, leading to the question:

Can system-call monitoring be made powerful enough to defeat current and future evasion attacks?

The answer to this question would be affirmative if we were to build a perfect system-call monitor that captures the totality of interactions between an application and its OS. Since these interactions define what we consider as an application's behavior, a monitor that can predict the totality of these interactions has to effectively duplicate the application logic. Moreover, in order to provide effective protection, the monitor shouldn't duplicate any of the application's vulnerabilities! Since this isn't a recipe for a practical system call monitor, we formulate the notion of a *I/O-data-oblivious* monitor that comes close to being perfect: the only difference is that such monitors ignore data buffer arguments to `read` and `write` system calls. This eliminates the need for the monitor to predict the application's behavior as a function of its inputs/outputs, thereby making it more practical. We note that existing system-call monitoring techniques [17, 20, 34, 30, 36, 25, 13, 27, 21, 32, 9, 15, 8, 11, 14, 5] are all I/O-data-oblivious[2].

Despite their power, we show that I/O-data-oblivious monitors can be defeated using an evasion attack that we call as *persistent interposition attacks.* This result implies that *current and future system call monitoring techniques, by themselves, are unlikely to can't provide a good defense against attack vectors that permit injected code execution.*

Overview of Persistent Interposition Attack. Like previous mimicry attacks, persistent interposition attack relies on code-injection vulnerabilities. It hooks into the victim application's execution path so that it can inspect and modify the arguments to all the program's `read` and `write` calls. For example, a persistent interposition attack could overwrite the global offset table (GOT) entries for the `read` and `write` functions in the C library. Our attack strategy is similar to a "man-in-the-middle" attack, where the attack code, in effect, modifies the input/output data of the victim application. Since a persistent interposition attack only modifies data read or written by the program, it is not powerful enough to obtain a root shell or gain a persistent foothold on the victim host. Nevertheless, many end-goals of today's attacks remain achievable:

[1]Henceforth, we use the term "IDS" to refer to system-call monitoring IDS.

[2]Content-based IDS, such as [35] aren't I/O-data-oblivious, but are vulnerable to attack techniques that are largely orthogonal to ours [10]. By combining the two techniques, these IDS can likely be evaded as well — See Section 5 for a more detailed discussion.

- *Steal credit-card numbers or passwords.* A persistent interposition attack against a web server processing e-commerce requests can enable an attacker to steal the credit-card numbers (or passwords) of customers visiting the server.
- *Redirect requests.* A hacker can use a persistent interposition attack against a domain's DNS server to redirect (some or all) visitors to the domain's web server to the hacker's web site. Alternatively, a DHCP server may be attacked in a way that enables an attacker to redirect clients to her name server, and subsequently, redirect all client requests to the destinations of her choice.
- *Impersonate a secure server.* A persistent interposition attack against a secure web server can allow an attacker to steal the server's private key, which would enable him to subsequently impersonate the server.
- *Suppress or alter emails.* An adversary can use a persistent interposition attack on a mail server to suppress or alter emails delivered by it.
- *Launch worms.* An attacker could use complementary attacks against a server and client to build a worm: compromised clients would infect servers and compromised servers would infect clients.

Persistent interposition is well-suited for attacking *long-running event-driven applications* that execute a request-response loop, e.g., most network servers, as well as some frequently targeted desktop applications such as browsers and email readers. In addition, the "embed-and-wait" strategy used in our attack can achieve more powerful end-goals against weaker IDS: if the IDS doesn't monitor system-call arguments, then our attack code can simply interpose itself on `open` and `write` operations, and when they are invoked, modify the arguments to create (or overwrite) a file of attacker's choice.

Previous Work in Mimicry Attacks. Mimicry attacks [33] can be defined as attacks that achieve attacker-intended effects without modifying aspects of an application behavior that are monitored by an IDS. The principal challenge is that of making them practical: showing that mimicry attacks can be crafted to exploit existing vulnerabilities in real-world applications, and can achieve typical goals of an attacker.

Prior mimicry attacks attempted to alter the control flow of their victims while generating system call sequences that would seem benign to an IDS, but this turned out to be quite complex. Typically, the attack code needs to make a large number of system calls, e.g., Wagner and Soto [33] found a sequence of 138 system calls that evaded the pH IDS while attacking `wuftpd`, while Gao et al [12] showed that the *shortest possible* mimicry attacks against gray-box IDS typically needed a few tens of system calls.

Modern gray-box IDS that inspect a program's runtime stack [9, 11] pose additional challenges for mimicry attacks. In particular, the attack code cannot *call* system calls since that would save the attack code's address on the stack, thereby revealing execution of code from a writable section of memory. Instead, it has to jump to some location in the application's existing code that invokes a system call. However, this means that control won't return back to the attack code after this system call, and hence the attack can't continue. To overcome this problem, Kruegel et al [22] devised a technique to corrupt memory locations (and registers) in a manner that ensures that control is returned back to the attack code when the code following the system call is executed. They developed an elegant technique to automate the identification of the memory locations that need to be corrupted, and the corresponding values. Their work showed that regaining control-flow is feasible, but didn't address several other issues that arise in constructing mimicry attacks against real-world applications:

- For complex applications, how feasible is it to set up the stack before each system call (invoked by the attack code) so as to escape detection by a stack-inspecting IDS?
- Note that typical attacks need tens of system calls, and each each call requires many operations to (a) set up its arguments, (b) set up the call stack, and (c) to modify memory locations needed for regaining control. Will the exploit code needed to accomplish these steps be small enough to fit within the size limits imposed by typical code injection vulnerabilities?
- Given that the technique of [22] successfully regains control only about 90% of the time, how feasible is it to string together the tens of system calls needed in typical mimicry attacks? Will repeated memory corruptions (needed for regaining control after each system call) cause a program to crash before the attack is complete?
- How easy is it to extend the attack technique to work against more powerful IDS, such as those that reason

about system-call arguments [21, 32, 14, 5]?

We discuss below how our work overcomes these challenges to develop working evasion attacks on real-world applications.

Advantages of Persistent Interposition Attacks. Our attack uses *persistent control-flow interposition*, rather than the *control-flow hijack* strategy used by previous mimicry attacks. As compared to control-flow hijack, we believe that our approach is *simpler, more reliable,* and *stealthy*, thus enabling us to develop a fully-working evasion attack against the Apache web server, and also demonstrate the necessary components of this attack on a few other servers. Our technique addresses some of the key issues raised above in developing practical mimicry attacks, e.g., restoring stack content after an attack, and limiting the size of attack code. It side-steps the other issues by employing a different attack strategy: *rather than eagerly attempting to control program behavior at every point, we take the lazy approach of lodging the attack at key places in the program* where such interposition can be done easily and reliably, and *without a need for repeated memory corruptions.* We show that this lazy approach can still achieve the attacker objectives outlined earlier.

To implement a persistent interposition attack, a hacker does not need to find a long sequence of acceptable system calls, which requires an understanding of control flows possible in a program. Instead, he only needs to find a convenient place to hook his code into the application's I/O routines. There are several easy choices, as described in Section 3. After this is done, *the victim application "cooperates" in the attack by invoking the attack code at convenient points during the request processing cycle of the victim application!* Moreover, data needed by the attack code is typically available at this point via parameters on the stack, thus avoiding the error-prone guessing of absolute memory locations of victim application's data structures.

By inserting itself into the request-processing flow of a server, *persistent interposition attack provides an active channel for the attacker to dynamically control and/or alter the behavior of the attack code.* For instance, an attacker can interactively examine and/or modify the memory of victim by sending appropriate "commands" embedded in legitimate-looking requests to the server. Alternatively, the attacker can upload new code that embodies additional attack capabilities onto the victim.

Note that the attacker is able to control the contents of victim's inputs remotely, and its outputs using the interposed attack code on the victim side. This makes it possible for the attacker to employ techniques such as those described in [10] to encode his requests and responses in a manner that blends it with normal traffic, making it possible to evade most content-based IDS using a persistent interposition attack.

In summary, persistent interposition attacks are quite convenient for implementing application-layer attacks. *They are practical for hackers to implement using skills and tools they already have, making them perhaps a more realistic and immediate threat, as compared to prior mimicry attacks.*

Paper Organization. Section 2 defines the class of *Input-Output data oblivious monitors*. The design of persistent interposition attack is described in Section 3, followed by its implementation and evaluation in Section 4. Implications of our attack on system-call monitoring techniques is discussed in Section 5, followed by a discussion of related work in Section 6, and concluding remarks in Section 7.

2 Input/Output-Data-Oblivious System-Call Monitors

In this section, we formalize I/O data-oblivious monitors that were intuitively described in the introduction. We also explain how they capture the likely limits of future system-call monitors.

We begin by formalizing the notion of program behaviors as observed by a system call monitor. In theory, a system call monitor could examine the entire memory of a monitored application at the point of invocation of each system call. For performance reasons, practical system call monitors limit the amount of memory that is examined to that of system-call arguments. In addition, gray-box anomaly detectors also examine the program's call stack. This suggests the following formalization of execution traces.

Definition 1 (Gray-box Execution Trace) *A gray-box execution trace (or simply, a* trace*) for a program P, denoted $T(P)$, is the sequence of all the system calls invoked by P during its execution. A system call in a trace provides information about its name, arguments (at the point of call and its return), return value, and the context in which it is made, i.e., the list of return addresses on the program's stack.*

Note that, in principle, a powerful monitor can incorporate knowledge about possible control-flows in the program, and hence can infer the calling context without explicitly reading the program memory. Nevertheless, we chose to incorporate the calling context in the definition in order to explicitly include so-called *gray-box anomaly detection* techniques [30, 9, 11] that rely on this information. Based on execution traces, program behaviors can be formalized as follows:

Definition 2 (Program Behavior) *The behavior of a program is the set $\mathcal{T}(P)$ of all traces generated by P during any of its legitimate executions.*

Not all of the possible executions of a program may be considered as acceptable uses of the program from a security perspective. We use the term "legitimate execution" in the above definition to eliminate unacceptable behaviors from consideration.

Practical system-call monitors accept a superset of $\mathcal{T}(P)$ defined above. However, one can imagine the extreme case where the monitor accepts exactly $\mathcal{T}(P)$:

Definition 3 (Perfect System-Call Monitor) *Given an observed execution trace T for a program P, a perfect monitor classifies T as legitimate if and only if $T \in \mathcal{T}(P)$.*

Since a perfect monitor has complete knowledge about the entire behavior of a program, it can potentially be used as a generator of traces rather than as an acceptor of traces. This is particularly easy to see in the case of a deterministic program P, since the next system call made by such a program is uniquely determined by its inputs until this point. Moreover, these inputs (including command-line and environment parameters) are fully captured by the system calls made thus far, together with their arguments and return values. Thus, a perfect monitor can uniquely determine the next system call that would be made by P from all the preceding calls. This suggests that such a monitor must effectively be duplicating the essential application logic contained in P. Moreover, it must do this without duplicating the vulnerabilities in P, or otherwise an attack may compromise the monitor as well. Practical monitors are unlikely to duplicate application logic, and hence won't be able to predict application behavior from its inputs/outputs. This observation motivates the following definition of practical system-call monitors.

Definition 4 (Input/Output Data Oblivious Monitor (IOM)) *An I/O-oblivious monitor accepts all traces T' obtained from any $T \in \mathcal{T}(P)$ by modifying the data argument to zero or more input and output system calls.*

Note that only the data-buffer arguments to system call that perform actual input/output, such as `read`, `write`, `send` and `recv` are being ignored; other arguments such as the file descriptors used, as well as the return value from the call are incorporated into the behavior model constructed by an IOM. Moreover, all arguments to every other system call, including I/O-related calls such as `open` are captured in the model.

We point out that IOMs are significantly more powerful than today's system-call monitors [25, 27, 34, 9, 24, 11, 5]. In fact, they are probably too powerful to be implementable now or in the future. Even so, we show that they can be evaded by our persistent interposition attack, which is designed to leave the stack unmodified at every system call, and only modifies the data buffer arguments to I/O system calls. As a result, persistent interposition attacks, by design, cannot be detected by any I/O data oblivious monitor.

We comment that although the intent of the IOM definition was to rule out very powerful monitors that could exactly predict application outputs (or more generally, its future actions) as a function of its inputs, it has the effect of ruling out more practical techniques such as those based on signature-based filtering or statistical profiling of input contents. However, other researchers [10] have already developed techniques that are orthogonal to ours in order to evade existing content-based IDS. These techniques rely on encoding attack inputs in a such a manner that their characteristics (e.g., byte frequency distribution) conform to the normal profile used by the IDS. These techniques can easily be combined with our attack technique since it gives the attacker full control over the contents of attack inputs as well as all outputs of the victim server.

3 Design of Persistent Interposition Attacks

In this section, we describe the design steps that are common across all persistent interposition attacks, and outline how these steps may be implemented for different applications. The specific choices made in our implementation are presented in Section 4.

Our starting point for persistent interposition attack is an exploit built on a typical code-injection vulnerability that enables execution of injected code without making additional system calls, e.g., a typical stack-smashing, heap-overflow or format-string vulnerability. Such an exploit may impose fairly stringent limitations in the size of exploit code, which may be much smaller than the code size needed to support sophisticated attacks, e.g., stealing of credit card information from a web server. For this reason, we design persistent interposition attack to proceed in three steps:

- *Initial exploit phase:* In this phase, a *bootstrapping* shim is inserted on the victim's normal execution path. Relatively small amount of code (about 100 bytes in our experiments) is needed to carry out the exploit phase, so that it can be used as a payload for most code-injection attacks.
- *Bootstrapping phase:* In this phase, additional attack code is sent by the attacker within legitimate-looking requests to the victim application. The bootstrapping code identifies these requests, assembles them into the code needed for the operational phase of the attack, and transfers control to the assembled code.
- *Operational phase:* Like the bootstrapping code, the operational code is also hooked into the victim's normal execution path. It examines every input and output, and modifies the outputs as desired by the attacker. The attacker may also upload additional code to change the attack over time.

Like previous works on mimicry attacks, our implementation assumes no defenses against memory corruption attacks. A defense such as address-space randomization (ASR) can make persistent interposition attacks harder to develop, but they will remain possible as long as there are code injection vulnerabilities. Specifically, with ASR, exploit code cannot hard-code the addresses of data or code objects that it wants to target. However, since most ASR techniques only randomize the base addresses of different memory regions, it is possible to develop scanning attacks to compute these addresses. Specifically, the exploit code can scan the stack for return addresses and data pointers. By comparing the addresses of corresponding objects (say, a specific global array or a return address pointing to a location within the executable) between victim and attacker's systems, it is possible to "derandomize" the locations of all objects within a memory region (i.e., global area or executable code area).

As noted in [22], mimicry attacks against stack-inspecting gray-box IDSs must not leave any trace of attacks on the call stack. In particular, the attack code needs to use a `jump` rather than a `call` so that its address won't be saved on the stack. This means that the attack code cannot get control when system calls (or any of the application's function calls) return, and hence it cannot immediately examine the data returned by a system call such as `read`. Instead, it has to wait for a subsequent function call that has been set up for interposition by the attack code. [22] develops a sophisticated static analysis to automate the identification of function pointers that could be hijacked for this purpose, and the same techniques could be applied here. However, in practice, we have found that persistent interposition attacks require very few functions to be interposed, and those can be easily identified by dynamically tracing the sequence of function calls made by the victim application. This is the approach we used in our implementation.

3.1 Phase I: Initial Exploit Phase

In this phase, persistent interposition attack uses a code-injection vulnerability to install the bootstrap code. It consists of the three steps described below.

Step I.1. Storing Bootstrap Code. The bootstrap code must persist long enough to upload the operational code, so we need to find a safe place to stow it. We considered the following candidate locations:

- *Stack.* If the victim application consumes only a limited amount of stack memory, the region of the stack beyond this space could be used by the attack. We need to ensure that the attempt to use this space does not cause a signal due to an invalid memory access. Some operating systems, including Linux, allocate stack space on demand and inspect the application stack pointer to determine whether a page-fault should be handled by extending the stack. To deal with this problem, our technique pushes a large amount of data onto the stack and then pops it off, and then copies the code into stack space allocated by the kernel as a result of these pushes.

- *Global buffers.* It is common for programs to use buffers that are much larger than the size of data likely to be stored in them. Alternatively, certain global arrays may hold rarely used data that can be overwritten without a significant chance of affecting program behavior. Note that the memory location of global variables is statically known, and hence the attack code knows the locations of such variables.
- *Heap.* Instead of global buffers, we can use heap-allocated buffers that contain rarely used data, or contain more memory than is typically needed. Pointers to such heap buffers are often stored in global variables, and so the exploit code can find them.

While one may need to choose among the three alternatives on other OSes, on Linux, the stack is likely to be the location of choice for storing bootstrap code: it can be used without significant risk of overwriting application data. Moreover, exploit-code doesn't need to know the locations of global or heap-allocated variables, which may be hard to obtain if defenses such as address-space randomization are deployed.

Step I.2. Interposing Bootstrap Code. After copying itself to a safe location, the initial exploit code modifies one or more pointers to functions that would be invoked during the victim's normal operation. To identify a suitable function pointer, the following choices need to be considered.

- *Application-specific support for plug-ins and modules:* Modern server and client software often provide extensibility features in the form of plug-ins or modules. Calls to module-provided functions (as well as some functions called by the module) are made using tables of function pointers. For instance, the Apache web server uses the `mod_ssl` module in order to support SSL. This module registers two functions with Apache that the web server can use to read and write encrypted data. These function pointers are an obvious target for a persistent interposition attack.
- *Virtual table pointers in C++ programs:* Virtual methods in C++ programs are implemented using a *vtable*, i.e. an array containing pointers to each virtual method implemented by the object's class. The bootstrap code can interpose on virtual method invocations by all objects in this class by overwriting these pointers in the vtable. This is particularly easy because C++ compilers usually generate one vtable, stored at a fixed location, for each class, and place a pointer to this table in every instance of that class[3].
- *Global Offset Table (GOT) entries:* The global offset table is used in Linux to dispatch calls made from the main executable to shared libraries, such as `glibc`, and from one shared library to another. Every externally-visible shared library function has an entry in the GOT. The dynamic linker maps the name of each shared library function into the corresponding address, and fills the corresponding entry. By default, Linux uses a lazy approach for resolving shared library functions, i.e., the address of a function is resolved the first time it is invoked. To support this, the GOT remains writable throughout the program's execution. Given this fact, and the fact that shared library functions are used by applications for performing all of their I/O operations, GOTs are an obvious choice for interposition — a fact that is leveraged commonly in attacks such as heap-overflow exploits, as well as previous mimicry attacks such as [22].

Once the target function pointers are identified, the exploit code modifies them to point to the bootstrap code.

Step I.3. Cleanup. The last step of the initial exploit phase is to cleanup any damage resulting from the exploit so that the victim application will continue executing without making any anomalous system calls. The technique for accomplishing this is dictated by the nature of the underlying vulnerability:

- *Heap overflow:* Heap overflow exploits typically overwrite a GOT-entry for a function that will normally be invoked by a victim application immediately after the overflow takes place. In this case, exploit code can resume normal execution by simply transferring control to the beginning of the original function called by the victim program. In our experiments described in Section 4.1, we were able to identify the location of the original function without any problem[4].

[3] Some compilers store vtables in read-only data section. To cope with this, we can modify the vtable pointer in a single object so that it points to a table constructed by the attack code. Typically, a suitable object can be identified by scanning the stack.

[4]It is usually quite easy to identify the location of the original function — the location of the function is usually identical between the attacker's machine and the victim machine. Even if there are differences, e.g., due to ASR, the relative distances between functions in the shared library will remain the same. As such, the exploit code can compute the location of the overwritten function from the location stored in the next GOT entry.

- *Stack-smashing vulnerability:* A typical stack smash results in the corruption of the return address in the stack frame containing the vulnerable buffer, and possibly some local variables of the caller's frame. Since the attack code gains control when the vulnerable function attempts to return to its caller, we only need to ensure that the caller continues normal execution. If the attacker can predict the values of the caller's corrupted local variables, then the attack can easily restore them. If some of the corrupted values can't be predicted, the attack code can return an error code to the caller, causing it to return early without using the corrupted variables. The attacker can evaluate these options on his system before settling on a specific choice.

  An alternative technique is to modify the exploit so that it does not write beyond the return address targeted by the attack. This may reduce the size of the payload, but still, most stack-smashing vulnerabilities involve moderate to large arrays, and hence can be expected to be sufficient to hold the initial exploit.
- *Format-string attacks:* Format-string attacks may overwrite a return address or a function pointer, such as an entry in the GOT. Typical format-string attacks cause a small amount of collateral damage, especially to the stack or the GOT. Possible techniques for recovering from such damage were already discussed above.

As the above discussion shows, heap overflows are especially suited for persistent interposition attack since the cleanup phase is very easy. In our implementation, we have also demonstrated a successful cleanup after a stack-smashing attack on Samba server.

3.2 Phase II: Bootstrapping Phase

Once the initial exploit phase is complete, the bootstrapping code is invoked during normal operation of a victim application. Typically, the bootstrapping code may be invoked on each `read` and `write` operation. After the bootstrap code finishes execution, it uses a jump to transfer control to `read` or `write` [5]. As pointed out earlier, it is important to use a jump (rather than a call) instruction to effect this control transfer, so that a system call monitor will not see any trace of the attack code on the stack. Specifically, all register values, including those of the stack pointer and base pointer, need to have the same values as at the invocation of the attack code.

Since a jump is used, control does not return to the bootstrap code after a `read` or `write`. In order to be able to read the data returned by `read`, the bootstrapping code uses the following strategy:

- Store the address of the buffer being used to read input data
- Intercept the next interceptable function call, and read and (optionally) modify the contents of the input buffer

Obvious choices for the subsequent interception are (a) the next system call invoked by the victim, (b) a `free` call that may be used by the victim to free a buffer used for reading the data, or (c) utility functions such as `strtok`, `strcpy` or `memcpy` that may be used to process the input data. In our experiments, we could readily identify the appropriate function call by examining the source code of the victim application, or by dynamically tracing the function calls made by it.

To upload additional code, the attacker sends specially marked inputs to the victim. The exact format of these inputs will need to be adapted to the victim application. Conceptually, these inputs contain (a) a marker that identifies a request from the attacker, (b) the code that is sent as part of this input, and (c) an op-code that indicates what the bootstrapping code should do with this code. When the marker is recognized by the bootstrapping code, it will copy the code into the memory region chosen in the previous step (or another free area of memory). Note that the marker could be implicit, e.g., the bootstrapping code may include logic that treats inputs from certain IP addresses as coming from the attacker. Alternatively, it could be a byte sequence that is explicitly included in the input. In this case, the attacker will aim to minimize the likelihood that other inputs accidentally contain the marker, but such accidents can be tolerated if they are rare: they would simply cause the current bootstrapping phase to fail, and require the attacker to retry the attack.

Note that the attack code contained in the input could be encoded in some way to reduce the likelihood that it would be identified by a content-based IDS. For instance, binary code may be encoded into ASCII data, and converted back by the bootstrapper.

The bootstrapping code should recognizes two op-codes: one that indicates a copy operation, and another

[5] Henceforth, when we refer to `read` and `write`, we are referring to all input and output system calls respectively.

that indicates the bootstrap code should transfer control to the beginning of the code uploaded during the bootstrap phase. The bootstrapping phase ends when the second opcode is processed.

3.3 Phase III: Operational Phase

The operational code uploaded during the bootstrapping phase performs the real work of the attack. Note that it is possible for this code to interpose on a different set of functions (as compared to the bootstrapping phase) by appropriately modifying function pointers in the process memory.

From now on, the operational code gets to see and modify all outgoing messages. It uses the same strategy as the bootstrapping code to examine input messages. The operational code can accomplish several attacker objectives using these two capabilities:

- *Extract client secrets.* An attacker can use a persistent interposition attack to obtain client credit card numbers or other personal data from an e-commerce web server.
- *Redirect clients.* The attacker can redirect clients to attacker-controlled hosts by modifying the responses to name-lookup queries, or even by modifying the target address of links in web pages served to them. Once clients have been redirected, the adversary can attack them using his server.

  Even if the redirection involves a cryptographically protected service such as HTTP over SSL, the attacker may be able to combine redirection with stealing of server's private key to carry out a successful attack.
- *Corrupt clients.* An attacker can use a subverted file-server to corrupt clients by giving them modified executable binaries. Alternatively, servers may compromise vulnerable clients by sending them malicious data, such as image or multimedia files that can exploit buffer overflows or other vulnerabilities in clients.
- *Drop messages.* A system call monitor may prevent the attack code from dropping an incoming message completely, but the attack can still alter the output actions that result from this input. For example, an attack on an email server could allow it to save a different email message in the file system from the one that was received.
- *Extract system secrets.* Many servers read in the system password file, `/etc/shadow`, to authenticate users. The attack code could embed the contents of this file in responses to the attacker, enabling her to perform an off-line dictionary attack.
- *Extract server secrets.* Servers that support the SSL protocol have a private key that is used to authenticate the server during connection negotiation. Since the key is used during every connection, most servers keep it in memory all the time. The attack code can look up the server key and embed it in the responses to the attacker's subsequent messages.
- *Extract arbitrary memory.* In general, the attack code could monitor incoming messages for commands of the form (a, n), and return the contents of n memory locations starting with the address a.

Standard root-shell attacks could accomplish these goals as well, but they would be detected by an I/O data oblivious IDS. Persistent interposition attacks show how to accomplish these goals while remaining stealthy.

4 Implementation and Evaluation

In order to focus our implementation efforts on aspects that are central to establishing practicality of persistent interposition attacks, we organized this section into three parts. In Section 4.1, we present a complete persistent interposition attack on the Apache web server. Section 4.2 considers the attack phases described in the previous section and evaluates the feasibility of different alternatives suggested for implementing them. In Section 4.3, we provide a theoretical rather than an empirical analysis of how persistent interposition attacks can be implemented on a few more server programs.

4.1 Apache Server with OpenSSL Vulnerability

OpenSSL versions before 0.9.6d contained a buffer overflow in the handling of client-provided keys, known as the KEY_ARG overflow. Solar Eclipse [1] developed a code-injection exploit against this overflow. This exploit overwrites the GOT entry for `free` with the address of the injected code. When the server subsequently calls `free`, it ends up executing the injected code that spawns a shell. We modified this exploit as described below to construct a persistent interposition attack.

4.1.1 Initial exploit phase

In our implementation, the initial exploit code was about 100 bytes, small enough to be accommodated in the payload of typical code-injection exploits.

To decide where to store the exploit code, we performed a dump of the global memory, and noticed that there were three large and mostly unused buffers: `ap_server_root`, `ap_server_confname`, and `ap_coredump_dir`. These character arrays are 8KB each, but the path names stored in them are typically only a few tens of bytes. We chose to copy our code to an offset of 100 bytes from the base of `ap_server_root`.

Apache includes an extensive plug-in architecture that enables dynamically loaded modules to override built-in functionality. The SSL extension to Apache overrides the basic input and output functions by registering two of its functions, `ssl_io_hook_read` and `ssl_io_hook_write`, as the read and write hooks. Our attack targets the `ssl_io_hook_write` function pointer for interposition. It saves the current value of this pointer and then overwrites it with a pointer to the attack code.

There are two benefits to interposing on the above two functions. First, it enables the attack code to access messages in plaintext rather than encrypted form. Second, the attack will continue to work *even if GOT were made read-only.*

It turned out that we did not need to interpose `ssl_io_hook_read` at all, since the data returned by a read operation is made available by Apache in a client request argument to `ssl_io_hook_write`. We relied on this fact to implement the attack entirely by interposing just a single function call, namely, `ssl_io_hook_write`.

The initial exploit overwrites the GOT entry corresponding to `free`. Thus, the cleanup phase in our attack consisted simply of restoring this GOT entry to point to the location of `free` function in `glibc`. Since the shared libraries were loaded at the same address on the victim and the attacker's machine, it was easy to predict the value needed for restoration.

4.1.2 Bootstrapping Phase

The bootstrapping code installed by the initial exploit phase now intercepts calls to `ssl_io_hook_write` and can inspect and change its arguments. As mentioned previously, an argument passed to this function also contains the data read from the client. The interposed code checks the input buffer for a special opcode indicating that a message contains the operational attack code. In this case, it copies the incoming operational attack code into the `ap_server_root`, while being careful to avoid overwriting itself. If the opcode indicates the end of operational code, then the bootstrap code updates the write hook to point to the operational code.

4.1.3 Operational phase.

The operational code uploaded by the attacker during the bootstrap phase could be very large in principle, and could perform the attacks described in Section 3. However, since the focus of our evaluation was on the initial exploit and bootstrap phases, we could do with a code size of about 200 bytes for a proof-of-concept operational phase. Our operational code simply monitored the number of requests handled by the compromised server. The attacker can query this number by sending specially crafted requests with an opcode recognized by the operational code.

Apache forks off several child processes to handle incoming requests, which poses some challenges to the operational phase of our attack. A successful completion of this attack compromises only one of these children. If subsequent attacker command packets are processed by a different child process, then our attack code will not see them. This problem is overcome either by repeating the attack to compromise multiple children, or by resending command messages until they reach the compromised child. A related problem is that Apache dynamically adjusts the number of running servers based on the number of incoming requests. If server load drops for a long period, Apache may kill some children processes. If all the compromised children are killed as a result, then the attacker needs to repeat the attack in order to compromise another one of the child processes that are still alive.

In summary, we successfully implemented a persistent interposition attack on the Apache server and were able to utilize the operational code to query the number of requests Apache had processed.

4.1.4 Verifying evasion of an I/O-data-oblivious monitor

By design, persistent interposition attacks aren't detectable by IOMs. Nevertheless, it would be useful to experimentally verify this. However, a direct verification, i.e., running the attack on an application monitored by an IOM, is not feasible since no implementations of IOM are available today (or are likely to be available in the future). Consequently, we need to rely primarily on manual reasoning based on the definition of IOMs to establish that an attack can't be detected by an IOM. Specifically, we used the following combination of manual reasoning and experimentation to verify that the attack presented above won't be detected by an IOM.

We used `strace` to log all system calls made by Apache. The system calls made by different children are logged into separate files using a command-line option provided by `strace`. First, we started the Apache server, and used a client to carry out the above attack, and logged the system calls. We then restarted the Apache server, and used the same client to send benign requests. We recorded the system calls in this case. These two steps were repeated several times to obtain multiple logs, each corresponding to one (benign or attack) run. We then used `diff` to compare the `strace` logs for each run. We observed that across the benign runs, there were small differences in the logs, such as the the position of `sbrk` and `mmap` calls (both used for memory allocation), file descriptor numbers and process ids. Since this variation is present in benign runs, an IOM, by definition, must accept these variations. We then compared benign runs with attack runs and verified that the differences between the two runs were the same as those observed between benign runs, or were due to the data arguments to `read` and `write` calls. Thus, an IOM would accept the attack run as well.

4.2 Implementing Persistent Interposition Attack on Other Applications

In this section, we consider each of the steps in persistent interposition attacks, and evaluate the ease with which they can be implemented. We chose a collection of applications (rather than a single one) for this evaluation, so that we can independently evaluate each of the implementation choices mentioned in Section 3.

4.2.1 Initial Exploit Phase

Storing Bootstrap Code. Several alternatives were discussed in Section 3 for this step. Of these, the feasibility of storing data in global buffers was already established by the Apache case study. We also verified the feasibility of copying attack data into the stack. We used a test program for this verification, but the details won't change across applications. Our implementation pushes data on the stack within a loop, and pops off this data. To ensure that the attack code won't be clobbered by the victim during its normal operation, the amount of data pushed should be more than the total of the maximum stack ever used by the victim and the memory needed by the attack code. Since our implementation could allocate hundreds of MBs of space in this manner without triggering any system calls, we did not pursue storage of attack data on the heap.

Interposing Bootstrap Code. Our Apache implementation demonstrated the feasibility of interposing on application-specific function pointers used to support module and plug-in functionality. Interposing on GOT is perhaps most convenient, since it works reliably across all applications in practice. We used interposition on GOT entries for two server programs, namely, `bind` and `lsh`. Interposing on virtual functions (or pointers to virtual function tables) is also likely to be quite easy. However, we did not have access to working exploits on real-world C++ applications, and hence didn't evaluate this choice experimentally.

Cleanup. Three common exploit types are prevalent today: stack-smashing, heap-overflow and format-string attacks. We obtained working exploits for the first two types, but could not find a fully-working exploit of a format-string vulnerability. Nevertheless, as described before, the technique for recovery from a format-string exploit is similar to that of heap-overflow or stack-smashing attack, depending on the nature of "collateral damage" resulting from a format-string attack.

Our Apache case-study has already demonstrated the ease of the clean-up phase on a heap-overflow exploit, so we consider stack-smashing exploits here. Specifically, we examined the `trans2open` vulnerability in the Samba server. This attack overflows a 1024-byte buffer on the stack, overwriting the return address on the stack frame above that of call_trans2open. We used an available exploit that launches a shell when trans2open returns to its caller. As with any stack-smashing attack, the value of the saved ebp pointer and the return address on the stack are modified, causing the victim to crash when the shell-code finishes. We modified this attack so that the process could recover. Specifically, the length of the overflow was reduced so as to avoid clobbering the local variables of the caller. Next, as soon as control was transferred to the attack code, it computed the expected value of ebp register from the value of the esp register and restored this value. It then executed a `ret` instruction. These changes added about 20 bytes to the shell-code and were sufficient to allow the samba process to continue normal execution after the attack without making any additional system calls.

4.2.2 Bootstrap Phase

The bootstrap phase remains essentially the same across all applications, so we did not pursue additional feasibility evaluation of this step beyond the Apache server.

4.2.3 Operational Phase

Since our intent is to demonstrate what can be accomplished by interposing attack code during normal operation of a victim, we simplified our evaluation task by interposing at the source-code level rather than using the more labor-intensive binary-code interposition. We ensured that we used only those capabilities that were available to code that would be injected in binary form into a working process, e.g., the ability to examine and alter input parameters to an interposed function, or to change global data. Using this approach, we verified that `bind` (a popular DNS server) and `lsh` (a GNU implementation of the ssh version 2 protocol) could be successfully attacked using a persistent interposition attack. We describe these results below.

BIND. DNS is a lightweight, connectionless, query-response protocol, so most DNS servers use a single process but may be multi-threaded. We examined `bind-8.2.2_p5` for targets for code interposition. The GOT entries for `sendto` and `recvfrom` are very convenient targets for an interposition attack. Specifically, our attack worked by interposing on `sendto`. Using documented DNS record formats, we were able to identify the location of the IP address (which is the most important piece of data within the DNS response) within the buffer argument to `sendto`. We simply modified this data to redirect clients to our server. It is possible to make this attack stealthier by doing this substitution selectively, e.g., when the query is from a certain IP address.

LSH. We inspected `lsh-1.4.2` for interposition targets. After authenticating a user, `lshd` does an `execve` on the `lsh-execuv` program with command-line arguments that specify the user-id and the group-id for the shell to be spawned. It is this `lsh-execuv` program that executes the actual `setuid` and `setgid` calls. In our attack, we targeted the GOT entry of `execve`. It was simple to modify the argument data at the entry of the `execve` call so as to set the userid to an arbitrary, attacker-chosen value. Depending upon the configuration of the LSH server, the attacker may still not be able to get a root shell this way, but he/she can easily assume any other userid. (Note that although this attack involves changing a system-call argument, the change is undetectable to an IDS, in the sense that the new value that we use would be a valid value in a different run of the server, where the user corresponding to this userid authenticated herself. The only difference between such a run, corresponding to valid authentication, and the attack sequence, is the authentication data itself, which appears as the data argument to an input system call — a difference that, by definition, can't be detected by an I/O data oblivious monitor.)

4.3 Attacking other servers

In this section, we discuss (rather than experimentally verify) possible persistent interposition attacks on a few additional servers.

DHCP Servers. On most networks, DHCP servers provide clients with their IP address, name server, and IP gateway, and may be configured to supply even more configuration parameters. An attacker could use a compromised DHCP server to redirect client DNS requests to a server under his control, or to redirect client packets through his computer, acting as a gateway.

The Internet Systems Consortium DHCP server handles all requests in one, long-lived process, making a persistent interposition attack against the DHCP server relatively easy and powerful.

Sendmail. Sendmail forks a new process to handle each incoming mail message, so a persistent interposition attack against sendmail's message reception code will be of limited use – the compromised server will exit soon, anyway. Sendmail can also forward messages, though, and, in its default configuration, uses a single process to handle all the messages queued up for later transmission. If an attacker finds a bug in sendmail's message forwarding routines, he can mount a persistent interposition attack on the forwarding process by inserting malformed messages in the forwarding queue. The attacker's code can then read and modify all subsequent messages processed from the same queue. This enables attackers to alter forwarded emails, misdirect them, or read the emails intended for arbitrary users.

5 Implications for Existing Defenses

System-call Learning Based IDS. Our work was motivated by the long series of research works in this area [17, 30, 36, 9, 11, 5, 21, 32]. All of them use I/O data oblivious models, and hence aren't able to detect persistent interposition attacks. The range of objectives that are achievable using persistent interposition attack suggests that one shouldn't rely on these IDS to protect against injected code attacks. Nevertheless, these techniques can be used to detect various other types of attacks such as race conditions, temp file bugs, and so on. More generally, due to its reliance on learning, these techniques have the potential to detect attacks that involve unintended uses of an application that wasn't captured in training.

Static-Analysis Based IDS. A number of system call monitoring IDS have been based on static analysis of application code. Some of these techniques rely on source-code analysis [34], while others can operate on binaries [15, 8, 14]. They construct I/O data oblivious models, and are hence susceptible to our attack.

The main advantage of this class of techniques is that they don't produce false positives. This is because they rely on static analysis techniques that are *sound* with respect to the semantics of the programming language (namely, C or C++) of the victim application. As a result, these techniques will detect only those attacks that cause the victim's code to behave differently from the semantics specified by its language. Many kinds of attacks, including race condition attacks due to TOCTOU vulnerabilities, temp file attacks, and various types of injection attacks, do not involve any violation of the semantics assumed by the static analysis, and hence aren't be detected by these techniques. Memory corruption attacks do involve a violation of the language semantics and hence fall within the scope of these techniques. However, our techniques show that it is possible to modify typical memory corruption exploits so as to evade detection by these IDS. This discussion highlights the need for additional research to determine if there are classes of attacks that can be reliably detected using static analysis based system-call monitoring IDS.

Policy-Enforcement Techniques and Specification-Based IDS. Some specific security objectives could be achieved using appropriately configured and enforced policies on system calls made by applications, and our results do not dispute this fact. For instance, Systrace and SELinux policies[6] have the primary objective of limiting damage to system resources and other applications that can result when a protected application is compromised. Their policies do achieve this objective. However, a user of these defensive mechanisms may incorrectly assume these policies actually protect the victim application itself by making it very difficult to carry out attacks that provide any significant benefit to attackers. Our result shows that *it is practical to carry out successful attacks that subvert the logic of applications confined by policy-enforcement engines.*

Techniques for Checking Integrity of Control Transfers. Recently, techniques have been developed that are related to system call monitors, but go beyond it in that they monitor most control transfers rather than just system calls. Control-flow integrity [4] transforms binaries to introduce integrity checks on targets of control transfers before any jump or call. Such a technique can disrupt the initial control-hijack step of code injection attacks. PAID [24] is a hybrid approach that incorporates a limited degree of integrity checking into a system call monitor. In particular, it uses source-code transformation to insert `notify` system calls before each indirect function call to report the target of the indirect branch to a system call monitor. Although this was designed as a measure to resolve nondeterminism in the automata model extracted from the program by PAID, it has the

[6]SELinux policies aren't stated in terms of system calls, but are close enough for our purposes, so we discuss them together with system-call based techniques.

effect that the monitor will come to know about any jump into injected code residing in data segment.

Injected code attacks seem virtually impossible with these techniques, so persistent interposition attacks aren't effective against them. However, there are simpler attacks that can succeed against these techniques by simply modifying system call arguments. Chen et al [7] demonstrate several powerful memory corruption attacks that operate by corrupting only data values, while providing the attacker with capabilities similar to those of code injection attacks. Although PAID can detect some of these attacks by examining system call arguments, due to the difficulty of accurate data flow analysis in languages such as C, there will likely be plenty of opportunities to craft successful data attacks.

Defenses against memory corruption attacks. The results of this paper, together with the above discussion, reinforce the idea that effective defenses against memory corruption attacks need to focus on the corruption step itself, rather than attempting to contain the damage that follows memory corruption. The best defenses are provided by techniques that detect a memory error before it happens, such as those described in [26, 19, 29]. However, these techniques may cause performance or compatibility problems, in which case one might rely on techniques designed to detect memory error exploits such as address-space randomization [3, 6].

Network IDSs and Payload Anomaly Detection Techniques. Network IDSs, such as Snort and Bro, routinely scan the content of packets in search of known attack signatures. However, signatures aren't available for unknown exploits, and hence a successful persistent interposition attack can be crafted based on such exploits.

Several techniques have emerged recently that detect intrusions by identifying anomalies in protocol payload data [35, 23]. Such content-based intrusion detection systems have been based on statistical analysis of input requests to a server, and recognizing anomalies such as binary data, or data with other unusual characteristics. These systems fall outside our definition of I/O data obliviousness. However, other researchers [10] have already developed techniques that are orthogonal to ours in order to evade existing content-based IDS. These techniques rely on encoding attack inputs in a such a manner that their characteristics (e.g., byte frequency distribution) conform to the normal profile used by the IDS (e.g., PAYL [35]). These techniques can easily be combined with our attack technique since it gives the attacker full control over the contents of attack inputs as well as all outputs of the victim server.

6 Related Work

We limit our discussions in this section to mimicry attacks and other related work that hasn't previously been discussed in this paper. Wagner and Soto [33] pioneered the concept of mimicry attacks. They suggest several strategies for constructing mimicry attacks, but ultimately choose one that hijacks control-flow of the victim application, and executes a system call sequence that is consistent with the application model used by the IDS. The attacker's objectives could still be achieved by altering the arguments to these system calls, since the IDS they considered didn't monitor system-call arguments. All subsequent works on mimicry attacks [31, 12, 22, 16] have relied on this strategy that couples control-flow hijack with modifications to system-call arguments.

Wagner and Soto pose the problem of generating such an attack sequence as a finite-state automata intersection problem, and generated a mimicry attack consisting of over 100 system calls that achieves the effect desired by an initial exploit consisting of 8 system calls. However, they did not implement a working mimicry attack. This problem was addressed by Tan et al [31], but their focus was on black-box IDS. Ours is the first working mimicry attack against real-world applications protected by gray-box IDS.

Gao et al [12] coined the term *gray-box anomaly detector*, and developed an elegant framework that unified previously known system-call anomaly detectors [17, 30, 9, 36], and further generalized them. They evaluate these anomaly detectors in terms of their resistance to mimicry attacks. For programs such as `wu-ftpd` and Apache `httpd`, they show that the minimum possible length of mimicry attacks is between 5 and 50 system calls, with the sequence length increasing with the precision of models.

Recent research has targeted the two main problems in developing the kinds of mimicry attacks described above. First, manual generation of these mimicry attacks is hard, given that typical attack sequences consist of several tens to hundreds of system calls. Second, as described in the Introduction, it is difficult for the attack code to make a series of system calls against a gray-box IDS.

Giffin et al [16] formulate the problem of generating mimicry attack sequences as a model-checking problem. The input to the model checker includes a (manually developed) specification of OS behavior, the program behavior model used by the IDS, and a specification that characterizes an "unsafe" OS state desired by an attacker. By using the OS model, their technique can generate all possible mimicry attacks that achieve the OS state desired by the attacker, instead of being limited by an initial exploit that served as the starting point for previous works. Moreover, the OS model enables the generation of arguments to system calls used in a mimicry attack. Although their formulation can handle push-down models, their evaluation considered the finite-state models generated by the Stide technique [17]. They did not consider gray-box IDS.

A significant problem in generating mimicry attacks against gray-box IDS is that system calls cannot be made directly by attack code, since the IDS can then detect the presence of a return address on the stack that falls outside of the program text. To cope with this problem, Gao et al [12] suggest that the attack code must jump to existing code in the victim application that will then make the system call on behalf of the application. However, this means that after the execution of system call, control will return back to the application rather than the attack code. To regain control, they suggest modification of a code pointer used by the application code following the system call so that it points back to the attack code. They showed the feasibility of this technique on a small example program, but manual development of mimicry attacks based on this technique for realistic programs poses a daunting challenge.

Kruegel et al [22] address the above challenge with a novel technique that automates the steps needed for regaining control. Specifically, they use symbolic code execution to compute relationships between the memory (and register) contents at the point where the attack code jumps into the application code, and the code pointers used subsequently by this code. By analyzing these relationships, their analysis identifies if control can be regained, and if so, the memory locations that need to be modified and their contents. They demonstrated their technique on three example programs (about 30 lines each), as well as real applications such as Apache, showing that about 90% of the time, control could be successfully returned to the attack code. However, the focus of their evaluation was to demonstrate the ability of their symbolic execution technique to generate configurations that can return control back to the attack code. As mentioned before, several additional problems that need to be addressed before constructing working mimicry attacks against real-world applications were left open.

7 Conclusion

It is well known that no intrusion detection system can precisely capture all deviations from an application's correct behavior, but our research shows that, with relatively little engineering effort, adversaries can execute powerful attacks while blending in almost undetectably with the sequences of system calls normally executed by an application. Our attack can evade all system-call-based intrusion detection systems with which we are familiar. The need to work against powerful system-call monitors that examine almost all system-call arguments will typically prevent persistent interposition attacks from achieving arbitrary goals such as gaining a root-shell, but we showed that typical end-goals of attackers such as stealing credit-cards or hijacking/impersonating servers can be achieved.

Whereas previous mimicry attacks required static analyses to discover system call sequences that can compromise an IDS, and to regain control between these system calls, our technique side-steps these problems by "co-opting" the vulnerable application into invoking the attack code at convenient points during its execution. As a result, persistent interposition attacks are practical for today's hackers to implement using skills and tools they already have, making them perhaps a more realistic threat as compared to prior mimicry attacks.

Persistent interposition attacks demonstrate the limits of system-call monitoring defenses in general, as any defense that could detect our attack would begin to emulate the monitored victim application. They call into question the feasibility of ever developing system-call monitors that can reliably detect the most common type of attack prevalent today, namely, code-injection attacks; and suggest that future research in system-call monitoring IDS should be focused on other classes of attacks.

Our results also highlight the importance of deploying dedicated defenses against powerful attack vectors such as memory errors, rather than relying on the secondary line of defense provided by intrusion detection systems. They also add to the body of evidence showing that control-flow monitoring has significant benefits over system-call monitoring.

References

[1] OpenSSL Vulnerability, `http://www.phreedom.org/solar/exploits/apache-openssl/`.

[2] Dyninst: An API for runtime code generation. `http://www.dyninst.org/`.

[3] PaX Project, The PaX team. http://pax.grsecurity.net.

[4] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. *ACM CCS*, 2005.

[5] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. *IEEE S&P*, 2006.

[6] S. Bhatkar, R. Sekar, and D.C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. *USENIX Security*, 2005.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. Non-control-data attacks are realistic threats. *USENIX Security*, 2005.

[8] H. Feng, J.T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. *IEEE S&P*, 2004.

[9] H. Feng, O. Kolesnikov, P. Folga, W. Lee, and W. Gong. Anomaly detection using call stack information. *IEEE S&P*, 2003.

[10] Polymorphic blending attacks. *USENIX Security*, 2006.

[11] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. *ACM CCS*, 2004.

[12] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. *USENIX Security*, 2004.

[13] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. *USENIX Security*, 2003.

[14] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection, *RAID*, 2005.

[15] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. *NDSS*, 2004.

[16] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. *RAID*, 2006.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. of Computer Security* , 1998.

[18] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. *USENIX Annual Technical Conference*, 2002.

[19] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *Int'l Workshop on Automated Debugging*, 1997.

[20] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. *ACSAC*, 1994.

[21] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. *ESORICS*, 2003.

[22] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. *USENIX Security*, 2005.

[23] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. *ACM CCS*, 2003.

[24] L. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. *Recent Advances in Intrusion Detection* , 2004.

[25] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. *FREENIX*, 2001.

[26] G. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. *POPL*, 2002.

[27] N. Provos. Improving host security with system call policies. *USENIX Security*, 2003.

[28] J. L. Rrushi and E. Rosti. Function call tracing attacks to kerberos 5. *DIMVA*, 2005.

[29] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. *NDSS*, 2004.

[30] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. *IEEE S&P*, 2001.

[31] K. Tan, K. Killourhy, and R. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. *RAID*, 2002.

[32] G. Tandon and P. Chan. Learning rules from system call arguments and sequences for anomaly detection. *Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

[33] D. Wagner and P. Soto. Mimicry attacks on host based IDS. *ACM CCS*, 2002.

[34] D. Wagner and D. Dean. Intrusion detection via static analysis. *IEEE S&P*, 2001.

[35] K. Wang and S. Stolfo. Anomalous payload-based network intrusion detection. *RAID*, 2004.

[36] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. *RAID*, 2000.