Secure Execution Via Program Shepherding

Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{vlk,iye,saman}@lcs.mit.edu

Abstract

We introduce *program shepherding*, a method for monitoring control flow transfers during program execution to enforce a security policy. Shepherding ensures that malicious code masquerading as data is never executed, thwarting a large class of security attacks. Shepherding can also enforce entry points as the only way to execute shared library code. Furthermore, shepherding guarantees that sandboxing checks around any type of program operation will never be bypassed. We have implemented these capabilities efficiently in a runtime system with minimal or no performance penalties. This system operates on unmodified native binaries, requires no special hardware or operating system support, and runs on existing IA-32 machines.

1 Introduction

The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program. This is done by exploiting bugs that allow overwriting stored program addresses with pointers to malicious code. Today's most prevalent attacks target buffer overflow and format string vulnerabilities. However, it is very difficult to prevent all exploits that allow address overwrites, as they are as varied as program bugs themselves. It is also unreasonable to try to stop malevolent writes to memory containing program addresses, because addresses are stored in many different places and are legitimately manipulated by the application.

Security attacks cannot be thwarted by simply inserting checks around application code that may cause system-wide changes. A malicious entity that gains control can simply inject its own code to perform any operation that the overall application has permission to do. Hijacking trusted applications such as web servers, mail transfer agents, and login servers, which are typically run with many global permissions, gives full access to machine resources.

Rather than attempt to stop a multitude of attack paths, where the protection is only as powerful as the weakest link, our approach is to prevent the execution of malicious code. We present *program shepherding* — monitoring control flow transfers to enforce a security policy. Program shepherding prevents execution of data or modified code and ensures that libraries are entered only through exported entry points. Instead of focusing on preventing memory corruption, we prevent the final step of an attack, the transfer of control to malevolent code. This allows thwarting a broad range of security exploits with a simple central system that can itself be easily made secure. Program shepherding also provides sandboxing that cannot be circumvented, allowing construction of customized security policies.

Program shepherding requires verifying every

This paper will be published in the Proceedings of the 11th USENIX Security Symposium (Security '02). This research was supported in part by the Defense Advanced Research Projects Agency under Grant F29601-01-2-0166.

branch instruction, which can be costly when done via instrumentation or in an interpreter. In order to reduce this overhead we perform security checks once and place the resulting trusted code in a cache, where it can be executed overhead-free in the future. Our implementation naturally fits within the RIO infrastructure, a dynamic optimizer built on the IA-32 version [3] of Dynamo [2]. Our system imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Although RIO is implemented for both Windows and Linux, this paper focuses on Linux only. We plan to extend our work to Windows.

In Section 2 we classify the types of security exploits that are prevented by program shepherding's three techniques, which are described in Section 3. Section 4 discusses methods of implementing program shepherding efficiently, and Section 5 describes the details of our implementation. Section 6 discusses how to prevent attacks directed at our system itself. We present experimental results and the performance of our system in Section 7.

2 Security Exploits

This section provides some background on the types of security exploits we are targeting. We classify security exploits based on three characteristics: the program vulnerability being exploited, the stored program address being overwritten, and the malicious code that is then executed.

2.1 Program Vulnerabilities

The two most-exploited classes of program bugs involve buffer overflows and format strings. Buffer overflow vulnerabilities are present when a buffer with weak or no bounds checking is populated with user supplied data. A trivial example is unsafe use of the C library functions `strcpy` or `gets`. This allows an attacker to corrupt adjacent structures containing program addresses, most often return addresses kept on the stack[7]. Buffer overflows affecting a regular data pointer can actually have a more disastrous effect by allowing a memory write to an arbitrary location on a subsequent use of that data pointer. One particular attack corrupts the fields of a double-linked free list kept in `malloc` headers [16]. On a subsequent call to `free`, the list update operation

```
    this->prev->next = this->next
```

will modify an arbitrary location with an arbitrary value.

Format string vulnerabilities also allow attackers to modify arbitrary memory locations with arbitrary values and often out-rank buffer overflows in recent security bulletins [6, 19]. A format string vulnerability occurs if the format string to a function from the `printf` family ({`,f,s,sn`}`printf`, `syslog`) is provided or constructed from data from an outside source. The most common case is when `printf(str)` is used instead of `printf("\%s",str)`. The first problem is that attackers may be able to read the memory contents of the process. The real danger, however, comes from the `%n` conversion specifier that writes back to the argument the number of characters printed so far. The location and the value of this number can easily be controlled by an attacker with type and width specifiers, and more than one write of an arbitrary value to an arbitrary address can be performed in a single intrusion.

In this paper we assume that attackers can exploit a vulnerability that gives them random write access to arbitrary addresses in the program address space. This ability can be used to overwrite any stored program address to transfer control of the process to the attacker.

2.2 Stored Program Addresses

Many entities participate in transferring control in a program execution. Compilers, linkers, loaders, runtime systems, and hand-crafted assembly code all have legitimate reasons to transfer control. Program addresses are credibly manipulated by most of these entities, e.g. dynamic loaders patch shared object functions, dynamic linkers update relocation tables; and language runtime systems modify dynamic dispatch tables. Generally, these program addresses are intermingled with and indistinguishable from data. In such an environment, preventing a control transfer to malicious code by stopping illegitimate memory writes is next to impossible. It requires the cooperation of numerous trusted and untrusted entities that need to check many different conditions and understand high-level semantics in a complex environment. The resulting protection is only as powerful as the weakest link.

Security exploits have attacked program addresses stored in many different places. Buffer overflow attacks target addresses adjacent to the vulnerable buffer. Stack allocated buffers allow the classic return address attack and a local function pointer attack. Heap buffer overflows also allow global function pointer attacks and a `setjmp` structure attack. Simple data pointer buffer overflows, `malloc` overflow attacks, and `%n` format string attacks are able to modify any stored program address in the vulnerable application — in addition to the aforementioned addresses, these attacks target entries in the `atexit` list, `.dtors` destructor routines, and in the Global Offset Table (GOT) [12] of shared object entries.

2.3 Malicious Code

An attacker can cause damage with injection of new malicious code or by malicious reuse of already present code. Usually the first approach is taken and the attack code is implemented as new native code that is injected in the program address space as data [20]. New code can be injected into various areas of the address space: in a stack buffer, heap buffer, static data segment, near heap, or even the Global Offset Table. Since normally there is no distinction between read and execute privileges for memory pages (this is the case for IA-32), the only requirement is that the pages are writable during the injection phase. Pointing any code pointer to the beginning of the introduced code will trigger intrusion when that pointer is used.

It is also possible to reuse existing code by changing a code pointer and constructing an activation record with suitable arguments. A simple but powerful attack reuses existing code by changing a function pointer to the C library function `system`, and arranges the first argument to be an arbitrary shell command to be run.

An attacker may be able to form higher level malicious code by introducing data carefully arranged as a chain of activation records, so that on return from each function execution continues in the next one [18]. A jump into the middle of an instruction (on IA-32 instructions are variable-sized) could cause execution of a malicious instruction stream, although this attack may be of very limited use.

3 Program Shepherding

The program shepherding approach to preventing execution of malicious code is to monitor all control transfers to ensure that each satisfies a given security policy. This allows us to ignore the complexities of various vulnerabilities and the difficulties in preventing illegitimate writes to stored program addresses. Instead, we can catch a large class of security attacks by preventing execution of malevolent code. We do this by employing three techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing.

3.1 Restricted Code Origins

In monitoring all code that is executed, each instruction's origins are checked against a security policy to see if it should be given execute privileges. For example, a policy could allow execution of code only if it is from the original application or library image on disk and is unmodified. The policy could allow dynamically generated code, but require that it execute within a layer of sandboxing. We describe in Section 5.1 how to distinguish original code from modified and possibly malicious code.

Restricted code origins alone can stop all security exploits that inject code masquerading as data into a program. This covers a majority of currently deployed security attacks, including the classic stack buffer overflow attack.

A hardware execute flag for memory pages can provide similar features to our restricted code origins. However, it cannot by itself duplicate program shepherding's features because it cannot stop inadvertent or malicious change to protection flags. Program shepherding uses un-circumventable sandboxing, described in Section 3.3, to prevent this from happening.

3.2 Restricted Control Transfers

Program shepherding enables security policies such as enforcing the calling convention by preventing return instructions from targeting non-call sites. Controlling return targets can severely restrict exploits that overwrite return addresses, as well as opportunities for stitching together fragments of existing code in an attack.

Another useful policy is restricting transitions from one segment to another, e.g. from application code to a shared library, or from one shared library to another. We can prevent malevolent jumps into the middle of library routines by restricting targets of calls and jumps to be on the library's export list and the source's import list.

3.3 Un-Circumventable Sandboxing

Sandboxing allows building customized security policies for different types of code. For example, checks can be added before loads and stores to ensure that only certain memory regions are accessed by application code.

With the ability to monitor all transfers of control, program shepherding is able to guarantee that sandboxing checks cannot be bypassed. Sandboxing without this guarantee can never provide true security — if an attack can gain control of the execution, it can jump straight to the sandboxed operation, bypassing the checks.

Sandboxing can provide detection of attacks that get past both restricted code origins and restricted control transfers. For example, an attack that overwrites a code pointer in order to call the `system` routine will not be stopped if `system` is allowed by the export and import lists. Program shepherding's guaranteed sandboxing can be used for intrusion detection for this and other attacks. The security policy must decide what to check for (for example, suspicious calls to system calls like `execve`) and what to do when an intrusion is actually detected. These issues are beyond the scope of this paper, but have been discussed elsewhere [15, 17].

4 Efficient Implementation of Program Shepherding

Our goal was to build an efficient system for monitoring control flow that runs on existing hardware and requires no modification to application source code or binaries. One possibility is instrumentation of application and library code prior to execution to add security checks around every branch instruction. However, this imposes significant performance penalties. Furthermore, an attacker aware of the instrumentation could design an attack to overwrite or bypass the checks.

Another possibility is to use an interpreter. Interpretation is a natural way to monitor program execution because every application operation is carried out by a central system in which security checks can be placed. Interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set. To reduce the emulation overhead, interpreters typically cache the native translations of frequently executed code so they can be directly executed in the future. By using a code

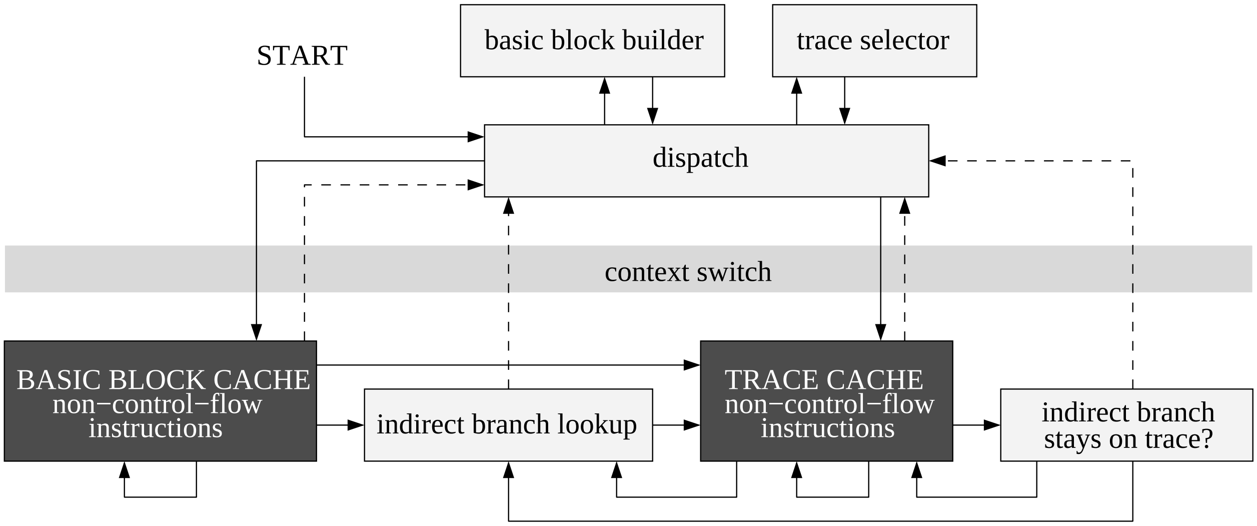

Figure 1: Flow chart of the RIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and RIO; application code and RIO code all runs in the same process and address space.

cache, we can perform security checks only once, when we copy the code to the cache. If the code cache is protected from malicious modification, future executions of the trusted cached code proceed with no security or emulation overhead.

4.1 Dynamic Optimization

A dynamic optimization system also utilizes this code cache design. We decided to build our program shepherding system as an extension to a dynamic optimizer called RIO. RIO is built on top of the IA-32 version [3] of Dynamo [2]. RIO's optimizations are still under development. However, this is not a hindrance for our security purposes, as its performance is already reasonable (see Section 7.5). RIO is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

A flow chart showing the operation of RIO is shown in Figure 1. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

Below we give an overview of RIO's operation, focusing on the aspects that are relevant to our implementation of program shepherding.

4.2 RIO: Runtime Introspection and Optimization

RIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block the application's machine state must be saved and control returned to RIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and is targeted via a direct branch, RIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup. Unfortunately indirect branch performance will never equal that of the original code, because a single instruction (the indirect branch) in the original execution has been expanded to multiple instructions.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [2, 23].

5 Implementation Details

This section discusses the implementation of the components of program shepherding discussed in Section 3. Most monitoring operations only need to be performed once, allowing us to achieve good performance in the steady-state of the program. In our implementation, a performance-critical inner loop will execute without a single additional instruction beyond the original application code.

5.1 Restricted Code Origins

The origins of a basic block are easily monitored by adding checks at the point where the system copies a basic block into the code cache. These checks need be executed only once for each basic block.

Code origins often require knowing whether code has been modified from its original image on disk, or whether it is dynamically generated. This is done by write-protecting all pages that are declared as containing code on program start-up. In normal ELF [12] binaries code pages are separate from data pages and are write-protected by default. Dynamically generated code is easily detected when the application tries to execute code from a writable page, while self-modifying code is detected by monitoring calls that un-protect code pages.

If code and data are allowed to share a page, we make a copy of the page, which we write-protect, and then unprotect the original page. The copy is then used as the source for basic blocks. If self-modifying code must be allowed, RIO keeps track of the origins of every block in the code cache, invalidating a block when its source page is modified. The original page must be kept write-protected to detect every modification to it. The performance overhead of this depends on how often writes are made to code pages, but we expect self-modifying code to be rare.

We handle new or modified code as specified by the security policy. We envision a series of protection levels, where original unmodified code is more trusted, and dynamically generated or modified code is less trusted, requiring additional sandboxing. Legitimate dynamically-generated code is usually used for performance; for example, many high-level languages employ *just-in-time compilation* [1, 11] to generate optimized pieces of code that will be executed natively rather than interpreted. This code almost always does not contain system calls or other potentially dangerous items. For this reason, imposing a strict security policy on dynamically-generated code (for example, disallowing the `execve` system call) is a reasonable approach.

5.2 Restricted Control Transfers

The dynamic optimization infrastructure makes monitoring control flow transfers very simple. For direct branches, any desired security checks can be performed at the point of basic block linking. If a transition between two blocks is disallowed by the security policy, they are not linked together. Instead, the direct branch is linked to a routine that announces or handles the security violation. These checks need only be performed once for each potential link. A link that is allowed becomes a direct jump with no overhead.

For an indirect branch, the hashtable lookup routine translates the target program address into a basic block entry address. Transitions between blocks using indirect branches are controlled by censoring the hashtable. We only place targets in the hashtable that are allowed by the security policy. A separate hashtable can be used for return instructions to ensure that they only target call sites. This separation has no effect on performance.

To require that all calls and jumps between segments satisfy the import and export lists, we can match targets against entry points of PLT-defined [12] or dynamically resolved symbols.

Security checks for indirect branches that only examine their targets have little performance overhead. However, examining the source and the target has the potential to slow down execution. This must be done either by adding explicit checks in the hashtable lookup routine, or by indexing the hashtable both by source and target.

5.3 Un-Circumventable Sandboxing

When required by the security policy, RIO inserts sandboxing into a basic block when it is copied to the code cache. In normal sandboxing, an attacker can jump to the middle of a block and bypass the inserted checks. RIO only allows control flow transfers to the top of basic blocks or traces in the code cache, preventing this.

An indirect branch that targets the middle of an existing block will miss in the indirect branch hashtable lookup, go back to RIO, and end up copying a new basic block into the code cache that will duplicate the bottom half of the existing block. The necessary checks will be added to the new block, and the block will only be entered from the top, ensuring we follow the security policy.

Page Type	RIO mode	Application mode
Application code	R	R
Application data	RW	RW
RIO code cache	RW	R (E)
RIO code	R (E)	R
RIO data	RW	R

Table 1: Privileges of each type of memory page belonging to the application process. R stands for Read, W for Write, and E for execute. We separate execute privileges here to make it clear what code is allowed by RIO to execute.

Restricted code cache entry points are crucial not just for building custom security policies with un-circumventable sandboxing, but also for enforcing the other shepherding features by protecting RIO. This is discussed in the next section.

6 Protecting RIO

Program shepherding could be defeated by attacking RIO's own data structures, including the code cache, which are in the same address space as the application. This section discusses how to prevent attacks on RIO. Since the core of RIO is a relatively small piece of code, we believe we can secure it and leave no loopholes for exploitation.

6.1 Memory Protection

To protect RIO we write-protect RIO's data and the code cache while control is in application code. We divide execution time into two modes: RIO mode and application mode. RIO mode corresponds to the top half of Figure 1. Application mode corresponds to the bottom half of Figure 1, the code cache and the RIO routines that are executed without performing a context switch back to RIO. We give each type of memory page the privileges shown in Table 1. RIO data includes the indirect branch hashtable and other data structures.

Initially, all application and RIO code pages are write-protected. When we enter RIO mode we unprotect the code cache and RIO data pages. If a basic block copied to the code cache contains a system call that may change page privileges, the call is sandboxed to prevent changes that violate Table 1. Program shepherding's uncircumventable sandboxing guarantees that these system call checks are executed. When we enter application mode we write-protect the code cache pages and RIO data pages. Because we do not allow application code to change these protections, we guarantee that RIO's state cannot be corrupted.

We protect RIO's Global Offset Table (GOT) [12] by binding all symbols on program startup and then write-protecting the GOT.

6.2 Multiple Application Threads

RIO's data structures and code cache are thread-private. Each thread has its own unique code cache and data structures. System calls that modify page privileges are checked against the data pages of all threads. When a thread enters RIO mode, only that thread's RIO data pages and code cache pages are unprotected.

A potential attack could occur while one thread is in RIO mode and another thread in application mode modifies the first thread's RIO data pages. We could solve this problem by forcing all threads to exit application mode when any one thread enters RIO mode. The performance cost of this solution would be minimal on a single processor or on a multiprocessor when every thread is spending most of its time executing in the code cache. However, the performance cost would be unreasonable on multiprocessor when threads are continuously context switching. We are still working on alternative solutions.

6.3 Interaction with Dynamic Optimization

We will maintain our security implementation as RIO is enhanced with classic compiler optimizations to improve performance. Some proposed optimizations maintain state while in application mode, requiring write permission on pages such that RIO cannot guarantee security. We plan to be involved in the design of future optimizations so that they can be incorporated securely into RIO.

7 Experimental Results

Our program shepherding implementation is able to detect and prevent a wide range of known security attacks. This section presents a test suite of exploits and then shows the performance of our system and the performance impact of our security techniques.

7.1 Test Suite

We constructed several programs exhibiting a full spectrum of buffer overflow and format string vulnerabilities. Our experiments also included the following applications with recently reported security vulnerabilities:

stunnel-3.21 CAN-2002-0002[8] A format string vulnerability in `stunnel` (SSL tunnel) allows remote malicious servers to execute arbitrary code because several `fdprintf` (a custom file descriptor wrapper of `fprintf`) calls have no format argument.

groff-1.16 CAN-2002-0003[8] The preprocessor of the `groff` formatting system has an exploitable buffer overflow which allows remote attackers to gain privileges via `lpd` in the `LPRng` printing system. The `pic` picture compiler from the `groff` package also has a format string vulnerability [21].

ssh-1.2.31 CVE-2001-0144[8] An integer-overflow bug in the CRC32 compensation attack detection code causes the SSH daemon (run typically as root) to create a hash table with size zero in response to long input. Later attempts to write values into the hash table provide attackers with random write access to memory.

sudo-1.6.1 CVE-2001-0279[8] `sudo` (superuser do) allows local users to gain root privileges. The vulnerability is triggered by long command line arguments and is caused by an out of bound access due to incomplete end of loop conditions. An exploit based on `malloc` corruption has been published [16].

Attack code is usually used to immediately give the attacker a root shell or to prepare the system for easy takeover by modifying system files. Hence, the exploits in our tests tried to either start a shell with the privilege of the running process, typically root, or to add a root entry into the `/etc/passwd` file. We based our exploits on several "cookbook" and proof-of-concept works [4, 26, 16, 21] to inject new code [20], reuse existing code in a single call, or reuse code in a chain of multiple calls [18]. Standard C library functions were used for existing code attacks. Chained calls were arranged by injecting carefully constructed activation records. On return from one function, execution continues in code in a function epilogue that shifts the stack pointer to the following activation record and continues execution in the next function of the chain.

Our test suite exploits were able to get control by modifying a wide variety of code pointers including return addresses; local and global function pointers; `setjmp` structures; and `atexit`, `.dtors`, and GOT [12] entries. We investigated attacks against RIO itself, e.g. overwriting RIO's GOT entry to allow malicious code to run in RIO mode, but could not come up with an attack that could bypass the protection mechanisms presented in Section 6.

All vulnerable programs were successfully exploited when run on a standard RedHat 7.2 Linux installation. Execution of the vulnerable binaries under RIO without security checks also allowed successful intrusions. Although RIO interfered with a few of the exploits due to changed addresses in the targets, it was trivial to modify the exploits to work under RIO.

Table 2 summarizes the contribution of each program shepherding technique toward stopping these attacks. We now describe these results in detail.

7.2 Restricted Code Origins

Enabling the code origin checks of RIO disallowed execution from address ranges other than the text pages of the binary and all mapped shared libraries. All exploits that introduce external code were detected and stopped.

A majority of currently deployed security attacks would be prevented by this technique alone. However, code origin checks are insufficient to thwart attacks that change a target address pointer to point to existing code in the program address space.

7.3 Restricted Control Transfers

We have evaluated which attacks would have been prevented by control transfer restrictions, which we are in the process of implementing.

Most of our vulnerable programs did not have any application code which could be maliciously used by an attacker. However, all of them had the standard C library mapped into their address space. Furthermore, many of the large programs imported all of the library routines that our attacks needed, so restrictions on cross-segment transitions would only stop a few of these attacks.

Requiring that return instructions target only call sites would thwart our chained call attack, even when the needed functions are explicitly imported and allowed by cross-segment restrictions. The chaining technique would be countered because of its reliance on return instructions: once to gain control at the end of each existing function, and once in the code to shift to the activation record for the next function call.

Note that if existing code used an indirect jump instruction to return instead of an actual return in-

Attack Type				Code Origins	Restricted Transfers	Sandboxing
Injected Code			Return Address	stops all	stops all	*policy dependent*
			Other Pointer			
Existing Code	Single Call	Imported	Return Address		stops most †	
			Other Pointer			stops `execve` ‡
		Not Imported			stops all	*policy dependent*
	Chained Calls				stops all	

Table 2: Capabilities of program shepherding against different attack classes.
†: Only code at a return point can be run.
‡: Since only a single call can be executed, sandboxing `execve` should prevent intrusion.

struction, our special return handling would be of no help. Such code will probably not be present in most applications — it will certainly not be generated by compilers since it breaks important hardware optimizations in modern IA-32 processors.

7.4 Un-Circumventable Sandboxing

Single malicious function calls to an imported library routine are still possible by modification of a function pointer, as are the simpler data-only attacks that only modify the argument of an otherwise valid function call.

We consider the readily available `execve` system call to be the most vulnerable point in a single-call attack. However, it is possible to construct an intrusion detection predicate to distinguish attacks from valid `execve` calls, and either terminate the application or drop privileges to limit the exposure.

7.5 Performance

Figure 2 shows the performance of RIO with and without program shepherding features. The figure shows normalized execution time on Linux for the SPEC2000 benchmarks [24] (compiled `-O3` and run with unlimited code cache space). The first bar gives the performance of RIO by itself. RIO's code layout optimizations enable it to speed up a number of the benchmarks. The second bar shows RIO's performance when it checks code origins to ensure that only unmodified, original code is executed. This overhead is negligible, as it occurs only at the point where basic blocks are copied into the code cache. The third bar gives the overhead of write-protecting RIO memory pages on every context switch. This overhead is again minimal, within the noise in our measurements for most benchmarks. Only `gcc` has significant slowdown due to page protection, because it consists of several short runs with little code re-use. We are working on improving our page protection scheme and completing implementation of the schemes for protecting RIO mentioned in Section 6 for multiple threads.

We are confident that the checks that are involved in restrictions on transitions between memory segments and on return targets will produce negligible overheads, as with the code origin checking that we have shown. We have implemented sandboxing of system calls, which introduces no noticeable overhead.

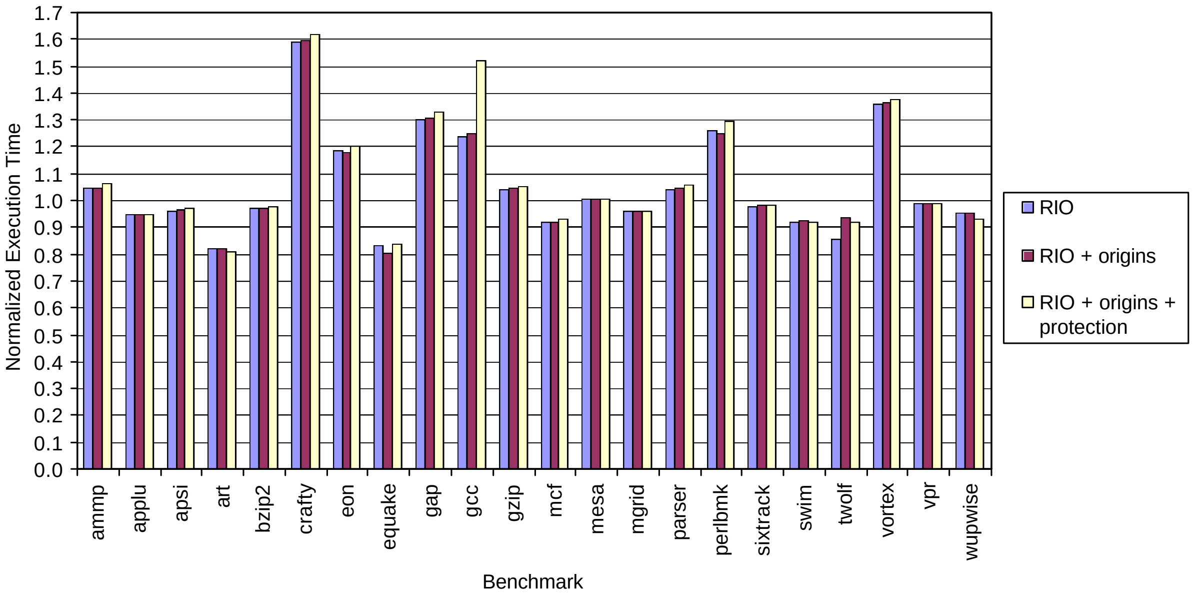

Figure 2: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [24] (excluding FORTRAN 90). The first bar is for RIO with no program shepherding implementation. The middle bar shows the overhead of checking code origins. The right bar shows the overhead of performing page protection calls to prevent attacks against the system itself.

8 Related Work

Reflecting the significance and popularity of buffer overflow and format string attacks, there have been several other works that attempted to provide automatic protection and detection of these vulnerabilities. We will shortly summarize the more successful ones.

StackGuard [7] is a compiler patch that modifies function prologues to place "canaries" adjacent to the return address pointer. A stack buffer overflow will modify the "canary" while overwriting the return pointer, and a check in the function epilogue can detect that condition. This technique is successful only against sequential overwrites and protects only the return address.

StackGhost [14] is an example of hardware-facilitated return address pointer protection. It is a kernel modification of OpenBSD that uses a Sparc architecture trap when a register window has to be written to or read from the stack, so it performs transparent operations on the return address before it is written to the stack on function entry and before control transfer on function exit.

Techniques for stack smashing protection by keeping copies of the actual return addresses in an area unaccessible to the application, are also proposed in the kernel modification in [14], and in the compiler patch StackShield [25] suffer from various complications in multi-threading environment and from deviations from a strict calling convention by `setjmp()` and exceptions. Unless the memory areas are unreadable to the application there is no hard guarantee that an attack targeted against a given protection scheme can be foiled. On the other hand, if the return stack copy is protected for the duration of a function execution, it has to be unprotected on each call and that can be prohibitively expensive (`mprotect` on Linux on x86 is 60–70 times more expensive than an empty function call). Techniques for write-protection of stack pages [7] have also shown significant performance penalties.

FormatGuard [6] is a library patch for eliminating format string vulnerabilities. It provides wrappers for the `printf` functions that count the number of arguments and match them to the specifiers. It is applicable only to functions that use the standard library functions directly, and it requires recompilation.

Enforcing non-executable permissions on the IA-32 via kernel patches was made for the stack pages in [10] and on all data pages with PaX [22]. Both provide no protection against attacks using existing code. Furthermore, our system provides execution protection from user mode and achieves better performance for protecting all data pages.

The system infrastructure itself is a dynamic optimization system based on the IA-32 version [3] of Dynamo [2]. Other software dynamic optimizers are Wiggins/Redstone [9], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on, and Mojo [5], which targets Windows NT running on IA-32. None of these has been used for anything other than optimization.

9 Conclusions

This paper introduces program shepherding, which employs the techniques of restricted code origins, restricted control transfers, and un-circumventable sandboxing to provide strong security guarantees. We have implemented program shepherding in the RIO runtime system and have shown that it successfully prevents a wide range of security attacks efficiently.

RIO does not rely on hardware, operating system, or compiler support, and operates on unmodified binaries on a generic Linux IA-32 platform. By performing security checks once and caching trusted code, our program shepherding implementation has minimal overhead.

We are expanding the list of security checks that shepherding can provide without loss of performance. We are also maintaining our security implementation with updates to RIO that improve performance.

Program shepherding allows operating system services to be moved to more efficient user-level libraries. For example, in the exokernel [13] operating system, the usual operating system abstractions are provided by unprivileged libraries, giving efficient control of system resources to user code. Program shepherding can enforce unique entry points in these libraries, enabling the exokernel to provide its better performance without sacrificing security.

We believe that program shepherding will be an integral part of future security systems. It is relatively simple to implement, has little or no performance penalty, and can coexist with existing operating systems, applications, and hardware. Many other security components can be built on top of the un-circumventable sandboxing provided by program shepherding. Program shepherding provides useful security guarantees that drastically reduces the potential damage from attacks.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[3] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[4] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[5] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[6] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities, 2001. In 10th USENIX Security Symposium, Washington, D.C., August 2001.

[7] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[8] Common vulnerabilities and exposures. MITRE Corporation. http://cve.mitre.org/.

[9] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, August 1999.

[10] Solar Designer. Non-executable user stack. http://www.openwall.com/linux/.

[11] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, January 1984.

[12] Executable and Linking Format (ELF). Tool Interface Standards Committee, May 1995.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level r esource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[14] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[15] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, Ca., 1996.

[16] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 8(57), August 2001.

[17] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proc. 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[18] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 4(58), December 2001.

[19] Tim Newsham. Format string attacks. Guardent, Inc., September 2000. http://www.guardent.com/docs/FormatString.PDF.

[20] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[21] Zenith Parsec. Remote linux groff exploitation via lpd vulnerability. http://www.securityfocus.com/bid/3103.

[22] PaX Team. Non executable data pages. http://pageexec.virtualave.net/pageexec.txt.

[23] Eric Rotenberg, Steve Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, December 1996.

[24] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation. http://www.spec.org/osg/cpu2000/.

[25] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux. http://www.angelfire.com/sk/stackshield/.

[26] Rafal Wojtczuk. Defeating solar designer non-executable stack patch. http://www.securityfocus.com/archive/1/8470.