Process Firewalls: Protecting Processes During Resource Access

Hayawardh Vijayakumar
SIIS Lab
Department of CSE
The Pennsylvania State University
hvijay@cse.psu.edu

Joshua Schiffman
Advanced Micro Devices
Josh.Schiffman@amd.com

Trent Jaeger
SIIS Lab
Department of CSE
The Pennsylvania State University
tjaeger@cse.psu.edu

Abstract

Processes retrieve a variety of resources from the operating system in order to execute properly, but adversaries have several ways to trick processes into retrieving resources of the adversaries' choosing. Such *resource access attacks* use name resolution, race conditions, and/or ambiguities regarding which resources are controlled by adversaries, accounting for 5-10% of CVE entries over the last four years. programmers have found these attacks extremely hard to eliminate because resources are managed *externally to the program*, but the operating system does not provide a sufficiently rich system-call API to enable programs to block such attacks. In this paper, we present the Process Firewall, a kernel mechanism that protects processes in manner akin to a network firewall for the system-call interface. Because the Process Firewall only *protects* processes – rather than sandboxing them – it can examine their internal state to identify the protection rules necessary to block many of these attacks without the need for program modification or user configuration. We built a prototype Process Firewall for Linux demonstrating: (1) the prevention of several vulnerabilities, including two that were previously-unknown; (2) that this defense can be provided system-wide for less than 4% overhead in a variety of macrobenchmarks; and (3) that it can also improve program performance, shown by Apache handling 3-8% more requests when program resource access checks are replaced by Process Firewall rules. These results show that it is practical for the operating system to protect processes by preventing a variety of resource access attacks system-wide.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection—Access controls
General Terms Security
Keywords Resource Access Attacks, Protection

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

1. Introduction

Programmers write their programs with expectations about the system resources they will obtain from particular system calls. For example, when writing a web server, programmers may expect to restrict the files served to a particular subtree in the filesystem (e.g., DocumentRoot in Apache), to use only system-approved shared libraries, and to create new files only the web server can access, even in shared directories (e.g., `/tmp`). However, adversaries have found several methods to direct victims to resources chosen by the adversary instead. We call these types of attacks *resource access attacks*. As a group, resource access attacks are less common than those based on memory errors (e.g., buffer overflows) or web programming errors (e.g., cross-site scripting), but they account for a steady stream of around 10% of reported vulnerabilities in the CVE [13] database (Table 1).

As an example, consider a typical web server that serves web content to authenticated users. Modern access control limits processes to *least privilege* [36] permissions, based on the functional requirements of the running program, but the web server needs to access the files containing its web content and password files for authenticating users. However, the program instructions that request these two sets of files are distinct, so the web server should not access the password file when it expects to access web content or vice versa. Adversaries can take advantage of flexibility in namespace resolution, ambiguity in environment variables, and plain old programmer errors to redirect the web server to violate such requirements. Access control cannot prevent such attacks because it treats all of the web server's system calls equally.

On the one hand, it was thought that these resource access attacks could be prevented by better programmer practice and extended system-call APIs, but many vulnerabilities remain, even in several cases when the programmers use the extended APIs. Extended system-call APIs allow programmers to check the properties of files retrieved by pathnames, but these checks do not block all adversary threats. For example, Chari *et al.* [8] show that checks must be applied to each pathname component to prevent link following attacks, but `lstat` only checks whether the *last* component is a link. Several other resource access attacks depend on sys-

Attack Class	CWE class	CVE Count	
		<2007	2007-12
Untrusted Search Path	CWE-426	109	329
Untrusted Library Load	CWE-426	97	91
File/IPC squat	CWE-283	13	9
Directory Traversal	CWE-22	1057	1514
PHP File Inclusion	CWE-98	1112	1020
Link Following	CWE-59	480	357
TOCTTOU Races	CWE-362	17	14
Signal Races	CWE-479	9	1
% Total CVEs	-	**12.40%**	**9.41%**

Table 1: Resource access attack classes. For each attack class, we show its Common Weakness Enumeration (CWE [14]) number, and the number of reported vulnerabilities in the Common Vulnerability Exposure (CVE [13]) database.

tem knowledge for which no API exists. For example, a bug in the Debian installer (BID 17288) led to Apache module binaries being installed with insecure RUNPATH settings, which allowed insecure library loading.

On the other hand, researchers have also shown that system-only defenses to resource access attacks [5, 11, 17, 39] are limited [6, 17] or incur false positives because they lack an understanding of the process's internal state [7]. In recent work, researchers propose using *capabilities* [27] to control the access to resources per system call using information flow control [26, 46]. However, the effectiveness of such defenses again depends on programmers using capabilities correctly, and it is often difficult or impossible to express defenses in practice using information flow (e.g., Time-of-Check-to-Time-of-Use TOCTTOU races and signal races).

This paper presents the Process Firewall, a kernel security mechanism designed to block resource access attacks system-wide. The Process Firewall examines the processes' internal state to enforce *attack-specific invariants* on each system call. By enforcing invariants on each system call, the Process Firewall provides attack-specific mediation that is analogous to a network firewall [10] for the system-call interface. Recall that prior to the introduction of network firewalls, host processes were trusted to protect themselves from network attackers. However, misconfigurations and programmer errors led to the need for a layer of defense to control hosts' access to network resources. The network firewall does not guarantee the security of hosts because some risky network accesses may be allowed for functional reasons, but the network firewall can block attacks by limiting network access to approved processes and controlling how connections are constructed. The Process Firewall provides similar defenses for the system call interface, limiting the resources available to particular system calls based on the process's internal state. Using this approach, the Process Firewall is a single mechanism that unifies defenses for a variety of previously unrelated resource access attacks.

Because of differing goals and representations, the Process Firewall is *complementary to system-call interposition mechanisms* for sandboxing [21, 23, 34] and host intrusion detection [18, 19, 37], as well as access control in general. These mechanisms aim to *confine* malicious processes by mediating any operations that they request. These mechanisms cannot utilize the process's internal state, such as its stack memory, because a malicious process could forge such state to circumvent confinement. In contrast, the Process Firewall only aims to *protect* benign processes from accessing resources that are not appropriate for their current state. Thus, the Process Firewall is able to use internal process state, in addition to detailed information about system resources already available to it, to enforce attack-specific invariants. If a process is actually malicious, it can only invalidate its own protection. Thus, the Process Firewall is not an alternative to system-call interposition mechanisms for confinement, but instead protects processes by blocking access to system resources that fail attack-specific invariants.

Designing the Process Firewall presents several challenges. First, we need a precise way to express the attack-specific invariants for resource access attacks. We find that a small number of resource and process attributes are sufficient to implement strong defenses from such attacks. Second, the low latency of the system-call interface demands methods to check invariants efficiently. We carefully design the Process Firewall's invariant checking mechanism to avoid unnecessary processing time in extracting resource and/or process information and to only process invariants for attacks that are possible for each system call. Third, the Process Firewall must be practical for use on system-wide defenses. The Process Firewall supports both binary and interpreted programs. Fourth, the Process Firewall must be easy to use, requiring no program modification or additional user configuration. We describe a variety of methods for OS distributors to produce practical invariants in which they can manage or eliminate false positives.

In our experiments, we show that several types of attacks can be prevented, including two previously-unknown vulnerabilities. For example, we comprehensively defend the Joomla! content management system from PHP File Include attacks, and prevent a variety of Untrusted Library Load attacks. We find that the Process Firewall incurs less than 4% overhead on a variety of macrobenchmarks. The performance overhead on individual system-call processing varies from $<3\%$ for system calls not dealing with resource access to $<11\%$ for system calls that do. Surprisingly, the Process Firewall can actually *improve* program performance and security simultaneously, if program checks are replaced by equivalent Process Firewall rules, as demonstrated by Apache, which was able to handle 3-8% more requests.

The result of this work is the Process Firewall system implemented for the Linux kernel, which includes: (1) a firewall rule language for expressing attack-specific invariants using attributes of processes and system resources; (2) mostly-automated methods for installing optimized Process Firewall rule bases; and (3) rule processing mechanisms spe-

Safe Resource	Unsafe Resource	Attack Class	Process Context
Adversary Inaccessible *(High Integrity, High Secrecy)*	Adversary Accessible *(Low Integrity, Low Secrecy)*	Untrusted Search File/IPC Squat Untrusted Library PHP File Inclusion	Entrypoint
Adversary Accessible *(Low Integrity, Low Secrecy)*	Adversary Inaccessible *(High Integrity, High Secrecy)*	Link Following Directory Traversal	Entrypoint
Same as prev. "check"/"use"	Diff. from prev. "check"/"use"	TOCTTOU Races	Entrypoint + System-Call Trace
No signal (Blocked)	Adversary delivers signal	Non-reentrant Signal Handlers	System-Call Trace + In Signal Handler

Table 2: Resource access attacks are caused by adversaries forcing victim processes to access unsafe resources instead of safe resources. Also shown is necessary process context to determine when these attacks apply (see Section 4).

```
/* fail if file is a symbolic link */
   int open_no_symlink(char *fname)
   {
01  struct stat lbuf, buf;
02 int fd = 0;
03  lstat(fname, &lbuf);
04  if (S_ISLNK(lbuf.st_mode))
05    error("File is a symbolic link!");
06  fd = open(fname);
07  fstat(fd, &buf);
08  if ((buf.st_dev != lbuf.st_dev) ||
09       (buf.st_ino != lbuf.st_ino))
10    error("Race detected!");
11  lstat(fname, &lbuf);
12  if ((buf.st_dev != lbuf.st_dev) ||
13       (buf.st_ino != lbuf.st_ino))
14    error("Cryogenic sleep race!");
15  return fd;
   }
```

(a) Symbolic link check

```
/* ignore malicious env var  */

   int load_library()
   {
01  if ((uid != euid) || (gid != egid)) {
02    /* SUID binary */
03    unsetenv("LD_LIBRARY_PATH");
04    unsetenv("LD_PRELOAD");
05  }
06  path = build_search_path();
07  foreach path p {
08    fd = open(p/lib);
09    if (fd != -1) {
10      mmap(fd);
11      break;
      }
    }
   }
```

(b) Library search path check

Figure 1: Program code to defend against resource access attacks.

cialized for system calls. Experiments show that the Process Firewall can be deployed with no user input or program changes to prevent a variety of attacks system-wide with a low likelihood of false positives.

2. Problem Definition

Processes require many system resources in order to function effectively. Resources may be delivered to processes either synchronously (e.g., files in response to `open`) or asynchronously (e.g., signals in response to `sigaction`). In either case, adversaries have several means for exploiting flaws when processes retrieve resources. Table 2 shows some resource access attacks, which occur when adversaries direct victim processes to resources chosen by the adversary. Columns 1 and 2 in Table 2 show the properties of the *safe resource* expected by the victim and the properties of *unsafe resources* to which adversaries direct victims for these resource access attacks.

In general, there are two ways by which adversaries can realize such attacks. First, an adversary can control how resources are retrieved directly. For example, when a victim program wants to read from a file in `/tmp`, the adversary who has write access to `/tmp` can create a symbolic link to `/etc/passwd`. If the victim does not detect that the target of the symbolic link is a high-secrecy file, the victim may end up accessing and possibly leaking the high-secrecy password file, when it meant to access a low-secrecy temporary file. Second, the adversary can indirectly control the resource accessed by tricking the victim process into requesting a resource specified by the adversary. For example, in a Directory Traversal attack [15], an adversary may request the file `../../etc/passwd` from a webserver. If the webserver does not properly filter the untrusted input or validate the retrieved resource, it could again end up accessing the high-secrecy password file when it meant to access a low-secrecy user web page.

An important characteristic of resource access attacks is that *a resource that is unsafe for a particular victim context is safe for some other victim context*. In the above example, the webserver can access `/etc/passwd` legitimately when it wants to authenticate clients. However, it should not do so when serving a user web page. As a result, traditional access control is insufficient to prevent resource access attacks, because it assigns permissions to processes as a whole, not distinguishing between what is safe or unsafe for different victim context.

2.1 Challenges in Preventing Resource Access Attacks

Figure 1 shows sample program code that aims to prevent resource access attacks to highlight the challenges. To prevent these attacks, victim programs either: (1) retrieve and check resource properties (first example) or (2) restrict the name used to retrieve the resource (second example).

Link Traversal and TOCTTOU Races. Figure 1(a) shows code from a program trying to defend itself against symbolic link attacks. On line 3, an `lstat` checks if the file is a symbolic link. If not, on line 6, the file is opened to read using the `open` system call. However, there is a race condition possible between lines 3 and 6. A TOCTTOU attack [5, 28] can be successful if the adversary forces the victim to "use" a different file on line 6 than its previous "check" on line 3. Row 3, Columns 1 and 2 in Table 2 show that an unsafe resource for a successful TOCTTOU attack is different from the previous corresponding "check" or "use" call, whereas the safe resource that protects against the attack is the same as the corresponding call. An adversary scheduled by the OS to run after line 3 but before line 6 could change the file to a symbolic link. To defend against this, an `fstat` is performed on line 7, and the inode and device numbers that uniquely identify a file on a device are compared to the `lstat` to make sure the file checked is the one opened on lines 8 and 9. However, even this is insufficient. Olaf Kirch [12] showed a "cryogenic sleep" attack, in which an adversary could put a `setuid` process to sleep before line 6 but after line 3, and wait for the inode numbers to recycle, thus passing the checks on lines 8 and 9. To defend this, an additional `lstat` is done on line 11, and the inode and device numbers compared again. So long as the file remains open, the same inode number cannot be recycled. Finally, Chari *et al.* [8] showed that such similar checks

must be done for each pathname component, not only the final resource, and proposed a generalized `safe_open` function that allows following of links so long as the link points to an adversary's own files and not the victim's files. Unfortunately, `safe_open` is used by few programs currently (Postfix uses a weaker version) and has false negatives[1].

Untrusted Search Path. Figure 1(b) shows simplified code from `ld.so` when it loads a library. An adversary, when launching a `setuid` program, may supply malicious path values for the `LD_LIBRARY_PATH` environment variables, forcing a victim program to use untrusted libraries. In this example, an untrusted library load resource attack succeeds if the adversary forces the victim to access a low-integrity resource on line 8 where the victim program was expecting a high-integrity resource (as shown in Columns 1 and 2 of row 1 in Table 2). To prevent this, `ld.so` unsets such environment variables on lines 3 and 4, builds a search path on line 6, opens the library file on line 8, and maps it into the process address space on line 10. Unfortunately, there are a variety of other ways that adversaries can control names in search paths apart from environment variables – RUNPATHs in binaries, programmer bugs, and even dynamic linker bugs (e.g., CVE-2011-0536, CVE-2011-1658, Payer *et al.* [33]). It is difficult for programmers to anticipate and restrict all sources of adversarial names for library search paths. For example, a bug in the Debian installer (CVE-2006-1564) led to Apache module binaries being installed with insecure RUNPATH settings, which allowed insecure library loading.

2.2 Limitations of Prior Defenses

Prior defenses can be divided into two broad categories: (1) system-only defenses and (2) program defenses. First, system-only defenses require no program modifications and are deployed either as libraries or in-kernel defenses. System-only defenses have been proposed for TOCTTOU [5, 11, 17, 39, 44] and link following [8] attacks. However, system-only defenses in general are fundamentally limited because they do not take into account *process context*. Process context captures the process's intent and therefore the set of valid resources for the process's particular system call. For example, Cai *et al.* [7] proved that without this process context, all system-only TOCTTOU defenses are prone to false positives or negatives.

On the other hand, while program code defenses can limit resource access per system call depending on process context, programs lack sufficient visibility into the system to defend against resource access attacks. Since resources are managed outside the program in OS namespaces, programs need to query the OS to ensure that they access the correct resource. However, there are several difficulties in doing so. First, such program checks are *complicated* under the current system-call API. As one example, the system-call API that programs use for resource access is not atomic, leading to TOCTTOU races. There is no known race-free method to perform an `access-open` check in the current system call API [7]. As another example, Chari *et al.* [8] show that to defend link following attacks, programmers should perform at least four additional system calls per path component for each resource access. Second, as a consequence of requiring additional system calls, program defenses are also *inefficient*. For example, the Apache webserver documentation [2] recommends switching off resource access checks during web page file retrieval to improve performance. Thirdly, program checks are *incomplete*, because adversary accessibility[2] to resources is not sufficiently exposed to programs by the system-call API. The first two rows of Table 2 show that adversary accessibility is necessary to identify unsafe resources for some attacks. Currently, programs can query adversary accessibility for only UNIX discretionary access control (DAC) policies (e.g., using the `access` system call), but many UNIX systems now also enforce mandatory access control (MAC) policies (e.g., SELinux [31] and AppArmor [30]) that allow different adversary accessibility. Finally, many programmers are unaware of resource access attacks and fail to add checks altogether. All these factors have led to resource access attacks making up around 10% of CVE entries (Table 1).

Program defenses such as privilege separation [35] and namespace isolation (using `chroot`), capability systems [27, 43], and information flow systems [26, 46] enable customized permission enforcement per system call. However, such solutions are manually intensive because programmers must tailor their programs to block resource access attacks. More importantly, these solutions are not portable because different deployments may have different adversary accessibility (determined by the access control policy), therefore having the impractical requirement of rewriting programs for each distinct system deployment to provide effective protection. In addition, some of these defenses do not address temporal properties of processes, necessary to block TOCTTOU and signal races. Lastly, privilege separation that uses the current system-call API inherits challenges of inefficiency [29] and complexity.

In this work, we want to develop a mechanism that protects each system call from resource access attacks. Noting the incompleteness, complexity, and inefficiency of program code checks, our solution has the following goals: (1) must be capable of preventing instances of the resource access attacks in Table 2; (2) must require no programmer or user effort to be deployed; (3) must be possible to configure use-

[1] In safe_open, an adversary A can trick V into accessing another victim B's files through A's symbolic links

[2] A resource is adversary accessible if the OS access control policy grants an adversary of the current process permissions to the resource. In UNIX discretionary access control (DAC), an adversary is a user with a different UID (except `root`). A similar approach can be applied to calculate adversaries for other security models, such as SELinux mandatory access control (MAC) [40]. Write permissions to the resource lead to integrity attacks and read permissions to secrecy attacks.

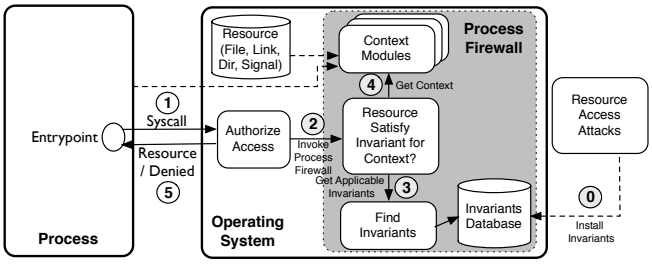

Figure 2: Process Firewall Design. Shaded steps are added by the Process Firewall.

ful policies that do not introduce false positives; and (4) must be more efficient than program-based defenses. That is, the Process Firewall must be an efficient and easy-to-use mechanism that prevents many resource access attacks without blocking valid function.

3. Solution Overview

Our solution is based on the insight that past defenses have been incomplete in stopping resource access attacks because they did not *simultaneously consider both the program request's particular requirements* and *system knowledge about adversary access to protect processes during resource access*. While system-only defenses do not consider the context of a program's particular request, program defenses do not have the system knowledge of adversary accessibility to resources. In this work, we propose a system-based solution, the *Process Firewall*, to protect programs against resource access attacks by augmenting the system's knowledge about adversary access with process context.

Our solution augments the OS kernel's authorization mechanism to account for the current process context and accessed resource context, such as adversary accessibility, as shown in Figure 2. First, during a setup phase (Step 0 in Figure 2), invariant rules (simply, *invariants*) are defined for attack types in Table 2 to form an invariant database. These invariants describe the preconditions, in terms of the process and resource contexts, under which a particular type of resource access attack is possible. While any process and resource context may be applied to an invariant in general, Table 2 shows that only a few of these (as explained in Section 4.2) are sufficient to block many resource access attacks. During runtime, when a process makes a system call, the OS authorization mechanism, a *reference monitor* [3], decides whether a subject responsible for the system call (e.g., the process) can perform the requested operations on the specified object (e.g., the resource) (Step 1 in Figure 2) using its access control policy. If the authorization mechanism allows a process access to the requested resource, the Process Firewall is invoked to block the resource access if it would result in any resource access attack (Step 2 in Figure 2). Step 3 of Figure 2 fetches invariants from the invariant database. Resource and process context to evaluate invariants is fetched using context modules (Step 4 in Figure 2). If the invariant precondition matches, the Process Firewall implements the specified action. In general, invariants are deny rules, where the default action is to allow the resource access (i.e., because no resource access attack was found to be possible in that process context for the target resource).

As an example, consider a `setuid root` binary vulnerable to an untrusted library search path attack (Figure 1(b)). Suppose the dynamic linker `ld.so` makes a system call on line 8 to load an adversary-accessible library. Since `root` processes are authorized to access any file, the authorization mechanism allows the process access to the library. The Process Firewall is then invoked (Step 2 in Figure 2). The Process Firewall fetches invariants from the invariant database (Step 3 in Figure 2). Assume that one invariant blocks the `open` system call in line 8 of `ld.so` from accessing adversary-accessible resources. To evaluate this invariant, context modules are invoked (Step 4 in Figure 2) to retrieve required process context (e.g., the process's user stack and the mapping of the file `ld.so` in the process) and resource context (e.g., that the resource is adversary-accessible) using the applicable context modules for the invariant's preconditions. In this case, the Process Firewall finds that the invariant precondition matches and thus blocks the resource access (Step 5 in Figure 2).

We identify a key difference between our goals and those of access control and host intrusion detection systems (IDSes) [18, 19, 37] that enable the deployment of such defenses in the kernel. Preventing resource access attacks only requires that we *protect* the process from unsafe resources. For both access control and host IDS systems, the goal is to confine a potentially malicious process, so they cannot act on any process context for fear of it being spoofed by the adversary. For example, host IDSes model the expected behavior of programs and compare the externally-visible behavior (e.g., process's system calls) to these models to detect intrusions. Unfortunately, they cannot trust any of the process's internal state because malicious processes can mimic a legitimate program [42]. In contrast, a malicious process that mimics another program to our system only affects its own protection, and access control still confines its operation. Section 4.4 describes how our mechanism protects itself from malicious or misbehaving processes during memory introspection.

4. Design

In this section, we describe the steps shown in Figure 2.

4.1 Defining Attack-Specific Invariants

To start (Step 0), we need to define what an attack-specific invariant is. These invariants define: (1) a *resource context*, which describes the properties of a resource that may indicate a resource access attack is in progress, (2) a *process con-*

text, which describes the properties of a process that would be vulnerable to the corresponding resource access attack implied by the resource context, and (3) an authorization decision to take when context matches (allow or block access). In Step 0 of Figure 2, known attacks, of the types summarized in Table 2, are translated into a database of attack-specific invariants. Invariants for attack types are either manually defined once across all deployments, or automatically generated for the specific deployment (Section 6.3). The challenge in this section is to define the format of the invariants necessary to prevent all these types of attacks. We find very few types of contextual information are necessary to express the necessary invariants for our current resource access attack classes: two for resource contexts and three for process contexts (Table 2).

Below, we define a Process Firewall *attack-specific invariant* as a function. Since Process Firewall attack-specific invariants augment access control, we will start with access control. An access control function takes as input subject label, an object label, and an operation, and returns whether access is allowed or denied.

$$\texttt{authorize}(\texttt{subject}, \texttt{object}, \texttt{op}) \mapsto \texttt{Y}|\texttt{N}$$

In contrast, a Process Firewall *attack-specific invariant* for preventing resource-access attacks augments the conventional authorization function by making decisions also depend on process and resource context:

$$\texttt{pf\_invariant}(\texttt{subject}, \textbf{entrypoint}, \textbf{syscall\_trace}, \texttt{object}, \textbf{resource\_id}, \textbf{adversary\_access}, \texttt{op}) \mapsto \texttt{Y}|\texttt{N}$$

Attacks are caused when an unsafe resource is returned instead of the safe resource for a particular process context. Column 2 in Table 2 shows that the required resource context to identify unsafe and safe resources are the *resource identifier* and *adversary accessibility*. Column 4 in Table 2 shows that the required process context is the program entrypoint and/or prior system calls executed by the process. Thus, to detect whether an invariant applies we may need to identify the program entrypoint and/or some part of the system call trace. Hence, in this case, a *subject*, *entrypoint* , and *syscall trace* identify the process context for detecting resource access attacks. An entrypoint is the program counter of a function call instruction on the process's call stack. For examples in Section 2.1, it can be thought of as identifying the line number of the function call in the program.

We use `pf_invariant` to block unsafe resources rather than allowing safe resources. This follows from our design decision to prevent false positives, at the cost of possibly allowing false negatives. By definition of the attack-specific invariants, any unsafe resource enables an exploit; therefore, there can be no false positives.

4.2 Checking Invariants

In Step 3, the Process Firewall's main function is retrieving process and resource contexts, and checking them against the invariants to verify that the resource request is not an attack. A naïve design simply fetches all process and resource contexts and then matches them against each invariant. Since context retrieval incurs overhead, we want to retrieve them only when necessary. Firstly, to prevent unnecessary context collection, we lazily retrieve context values. Secondly, to preserve and reuse gathered context as long as it is valid, we support module-specific caching.

Lazy context retrieval gathers context only when it is needed by an invariant. The Process Firewall associates each context field with a bit in a *context bit mask* that shows which context field values have already been collected. To enable modular context retrieval, we designed *context modules*. Each context module retrieves one context field value. When evaluating an invariant, the rule matching mechanism checks if all necessary context field bits are set. If not, it triggers the associated context module.

Computed context field values themselves are then cached for reuse. Once we have obtained and stored a context value, this value may apply across other rules and even across multiple invocations of the Process Firewall. For example, the process call stack used to find program entrypoints is valid throughout a single system call, but multiple resource requests may be made (e.g., in pathname resolution). Context modules must support an API to check whether to invalidate their context at the beginning or end of rule processing.

4.3 Finding Applicable Invariants

As the size of our invariant database grows, sequential evaluation becomes impractical. Several systems, such as Windows access control lists and network firewalls sequentially scan rules until a match is found, leading to long authorization times for large rule sets. To combat this problem, network firewalls provide facilities to organize rules into *chains*, enabling only applicable rules to be run. The problem is such chains are usually manually configured in network firewalls.

To solve the problem of sequential rule traversal and manual configuration, we automatically create chains for entrypoints. Because nearly all our invariants are associated with a specific entrypoint, we organize our invariants into *entrypoint-specific chains* and traverse the chain specific to an entrypoint. Thus, the Process Firewall determines the entrypoint associated with this resource request and traverses only that chain. Rules that do not involve entrypoints are matched before jumping to entrypoint-specific chains.

This simple traversal arrangement is possible because we have only deny rules (Section 4.1) followed by a default allow rule. If we had both deny and allow rules, then the order of traversal of rules would be important, and rule organization also would also become more complicated as in network firewalls.

4.4 Retrieving Entrypoint Context

In Step 4, we run the context modules necessary for the current invariant being checked, as described in Section 4.2. The

only context values that need to be retrieved from the processes' internal states are the entrypoint contexts. The Process Firewall is designed to protect processes from resource access attacks system-wide, so we need to be able to retrieve entrypoint contexts from multiple types of programs being run on the system. In this section, we explore how to safely retrieve entrypoint contexts for different types of programs.

Our entrypoint context modules should handle both binary and interpreted programs. For binaries, the challenge is to reason correctly about compiler optimizations, such as tail-call elimination, and compile-time options, such as those that remove frame-pointer information. In these cases, we can still retrieve the call stack if debug or exception handler information is available (e.g., Ubuntu by default compiles all programs with exception handler information). In case such information is unavailable, we fall back to producing a stack trace using function prologue information (e.g., as used by GDB). For interpreted programs[3], we adapt the backtrace code from the interpreter to run in the kernel. This is only a small amount of code, ranging from 11 lines for PHP to 59 lines for Bash. We have not found any programs in the Ubuntu 10.04 desktop distribution or LAMP stack for which the call stack is not retrieved correctly by these methods.

Since context modules obtain data from potentially malicious processes, they need to perform careful input sanitization to avoid arbitrary kernel compromise through invalid pointer dereferences or denial-of-service (DoS) attacks such as unwinding infinite call stacks. Our entrypoint context prevents invalid pointer dereferences by using the kernel's `copy_from_user` function. Further, to prevent DoS attacks, it sets an upper limit on the number of stack frames. These techniques ensure the Process Firewall aborts evaluation of malformed context without itself exiting or functioning incorrectly. As a result, an applicable rule may mistakenly not match for a malicious process, but this only affects the malicious process's protection.

5. Implementation

We have implemented versions of the Process Firewall for Linux kernel versions 2.6.35 and 3.2.0. We find that the Process Firewall performs a service similar to a network firewall. The Process Firewall controls access to specific resources (e.g., by resource identifier) or groups of resources (e.g., by label) by specific ports (e.g., entrypoints) in a stateful way (e.g., system-call traces). As a result, we construct the Process Firewall by adapting the `iptables` firewall mechanism. The main benefit we derive from the `iptables` architecture is the extensibility of the rule language and modules to new attacks, just as `iptables` is extensible to new protocols. Thus, we can extend the system-call API modularly to arbitrary process and resource contexts and new attacks without affecting core kernel code.

[3] We support the Bash, PHP, and Python languages thusfar.

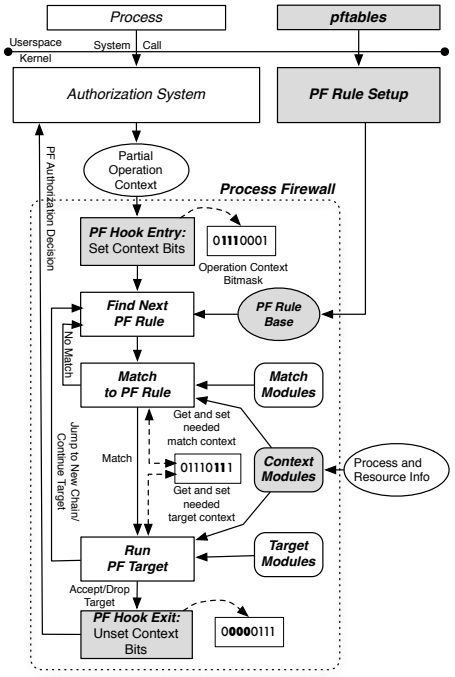

Figure 3: The Process Firewall implementation: Shaded elements are specially developed for Process Firewall rule installation and processing. Unshaded elements in the Process Firewall proper are standard firewall components.

Adding code to the kernel increases the TCB and also raises questions of maintainability. The Process Firewall core consists of 1102 LOC (501 LOC for rule traversal and matching, 378 LOC for validating and setting up rules pushed in from userspace into chains, and 223 LOC for initialization and module registration) with 2451 LOC for modules (dominated by the entrypoint context module with 1735 LOC). Bugs in interaction with userspace (validating rules and fetching entrypoints) may cause kernel compromise. However, the core code (including rule validation) is mostly borrowed from the mature `iptables`, and the entrypoint context module (modified from a proposed kernel patch [32]) carefully handles interactions with potentially malicious userspace as described in Section 4.4. Thus, we believe the increase in TCB is acceptable. Finally, since Process Firewall code is divorced from the mainline kernel code, it is easily maintainable. Porting from the Linux 2.6.35 to Linux 3.2.0 kernel required minimal effort.

5.1 Process Firewall Rule Processing Loop

Figure 3 shows an overview of the Process Firewall system, where the components added for the Process Firewall are shaded. When a user-level process submits a system call request, the kernel's existing authorization system (e.g., SELinux [31] over the Linux Security Modules interface [45]) uses the host's access control policy to authorize the set of security-sensitive operations resulting from each system call. If an operation is authorized, the Process Fire-

wall is then invoked through a *PF hook* to further determine whether the resource is appropriate for that particular process context based on the specified invariants. These invariants are stated in terms of firewall-style rules stored in the *PF Rule Base*. We use LSM for the Process Firewall rather than build on system-call interposition techniques [21, 34] as LSM has no race conditions [20] and has been analyzed for complete mediation to resources [25, 47].

Once invoked, the Process Firewall starts processing the first rule in the rule base. A rule matches a "packet" if all its classifiers (Table 3) match values in the "packet." A rule can contain several classifiers, and user-defined classifiers can be added through extensible *match modules*, similar to how `iptables` extensibly handles network protocols.

In a network firewall, the packet to match is readily available. The Process Firewall constructs its "packet" by fetching information required by match modules from the process and resource through context modules. Context collected is recorded using a bitmask. If a rule matches, *target modules* are invoked, which either produce a decision to be returned to the authorization system, ask to continue processing the next rule, or jump to a new chain of rules. This is again similar to `iptables`, where different target modules (called jumps) can accept or drop packets. If a rule is not matched, processing continues on the next rule.

Those wishing to write new match and target modules must write both a userspace part that handles rule-language extensions, and a kernel handler function, similar to `iptables`. For the Process Firewall, module writers also have to implement the necessary context modules to obtain required context for their match and target modules.

One critical issue is that `iptables` is not re-entrant – it saves the stack of chain traversals of a packet with a table. Thus, any re-entry invalidates this stack. To defend against this, `iptables` turns off kernel pre-emption and interrupts. However, we noticed that disabling interrupts on each resource request, perhaps several times per system call, has a noticeable impact on system interactivity. Further, we found that such disabling also leads to overhead in the performance-critical main loop.

To solve this issue, we designed the Process Firewall to run with interrupts enabled. Instead of saving the stack of rule traversals with the table, as `iptables` does, we instead maintain a per-process rule state, by extending `struct task_struct`. The process can thus be safely scheduled out during rule-base traversal.

5.2 Process Firewall Rule Language

Table 3 shows the rule language of the Process Firewall. This is analogous to `iptables`. There are statements that identify the entire firewall (`pftables`), tables, and chains (described above). Individual rules have the same structure as an `iptables` rule, consisting of default matches (`def_match`), custom matches (`match`), and targets. However, these rule elements are specific to the Process Firewall, representing the difference between the network firewall concepts and those used in the Process Firewall. For example, default matches specify the five context values: (1) process label; (2) resource label[4]; (3) resource identifier (signal or inode number); (4) program binary; and (5) entrypoint (`program` and `entry_point`). Entrypoint program counters are specified relative to program binary base, handling ASLR code randomization. The special keyword `SYSHIGH` denotes the set of all trusted computing base (TCB) subjects (for `-s`) or objects (for `-d`) for SELinux [24, 40]. Match and target modules in a rule can refer to a context in their arguments (e.g., C_INO for inode number); this is replaced by the actual context value at runtime.

Rule Language

```
pftables [-t table] [-I|-D] chain rule_spec
table : [filter | mangle]
chain : [input | output]
rule_spec : [def_match] [list of match] [target]
match : -m match_mod_name [match_mod_options]
target : -j target_mod_name [target_mod_options]
def_match : -s process_label -d object_label
          : -i entry_point -o lsm_operation -p program
```

Example Rule

* Disallow following links in temp filesystems.

```
pftables -t filter -o LNK_FILE_READ -d tmp_t -j DROP
```

Table 3: Process Firewall rule language with an example.

We have developed several match, target and context modules to handle the resource access attacks in Table 2. The `STATE` match and target modules allow matching and setting arbitrary key-value pairs in a process-specific (in Linux, also thread-specific) dictionary, implemented by extending the `struct task_struct`. This stores, for example, the resource identifier (inode number) accessed in previous system calls to defend TOCTTOU attacks, and if the process is currently handling a signal, to defend signal races. The `LOG` target module logs a variety of information about the current resource access in JSON format.

The Process Firewall rules are inserted into the kernel by the *pftables* user-space process. The *PF rule setup* module translates input rules into an enforceable form and organizes them for efficient access in the *PF rule base*. Each chain in the Process Firewall has rules that block unsafe resources (`-j DROP`) followed by a default allow policy (`-j ACCEPT`). In addition, it translates filenames into inode numbers and SELinux security labels into security IDs for fast matching.

6. Evaluation

In this section, we examine the Process Firewall's ability to block exploits, methods to generate rules for the Process Firewall in a manner that does not produce false positives, and Process Firewall performance.

[4] The SELinux MAC system assigns labels to processes (subject labels) and resources (object labels). For example, the process `sshd` has the label `sshd_t` while the file `/etc/shadow` has the label `shadow_t`. In SELinux, the relevant part of all subject and object labels are called types. The `_t` denotes a type.

#	Program	Reference	Class
E1	Apache	CVE-2006-1564	Untrusted Library
E2	dstat	CVE-2009-4081	Untrusted Search Path
E3	libdbus	CVE-2012-3524	Untrusted Search Path
E4	Joomla! gCalendar	CVE-2010-0972	PHP File Inclusion
E5	openssh	CVE-2006-5051	Signal Handler Race
E6	dbus-daemon	Unpatched	TOCTTOU
E7	java	Unpatched	Untrusted Search Path
E8	Icecat	Unknown	Untrusted Library
E9	init script	Unknown	Link following

Table 4: The exploits tested against the Process Firewall.

6.1 Security Evaluation

To evaluate the effectiveness of the Process Firewall in blocking resource access attacks, we deployed it on an Ubuntu 10.04 distribution and tried to exploit resource access attacks against programs. We tested 9 exploits shown in Table 4 that are representative instances of many resource access exploits. Four exploits (E1-E4) were chosen to check effectiveness of four rules that were automatically suggested by our rule suggestion procedure (Section 6.3). Note that these rules were suggested with no knowledge of the exploits we tested them against. E5 tested the manually-inserted, non-reentrant signal handler rules (R8-R11). E6, E7 were chosen to check effectiveness of rules that were automatically generated from known vulnerabilities, to externally protect unpatched programs. E8, E9 were new vulnerabilities automatically blocked by the Process Firewall itself. We verified that all exploits were successful when the Process Firewall was disabled. When enabled, the Process Firewall successfully blocked these resource access attacks.

6.1.1 Common Vulnerabilities

E1: Apache. CVE-2006-1564 is an untrusted library load vulnerability based on an insecure RUNPATH discussed in Section 2.1. To simulate this condition, we manually set RPATH to the insecure value. Rule R1 blocked this attack, as the SELinux label of `/tmp/svn` (`tmp_t`) was not in the set of valid labels for the `ld.so` entrypoint that opens library files. Section 2.1 listed many other reasons for untrusted library search paths – all are blocked by the single rule R1.

E2: dstat. dstat is a Python script that outputs a variety of performance statistics. It had an untrusted module search path (Python `os.path`) that included the working directory, enabling adversaries to plant a Trojan horse Python module. Rule R2 constrains Python scripts to load only trusted Python scripts labeled `usr_t, lib_t`, corresponding to `/usr/lib/` and `/usr/share` directories, which blocked this attack. Python programmers are often the cause for such bugs, but other reasons exist – in 2008, the Python interpreter itself set insecure search paths (CVE-2008-5983), affecting a variety of scripts. All such attacks are blocked by R2.

E3: libdbus. D-Bus is a message bus system that various applications use to communicate. Clients use `libdbus` to talk to the D-Bus server socket. However, `libdbus` programmers did not expect to be called from `setuid` binaries, so they did not filter an environment variable that specifies the path of the D-Bus system-wide socket. This is a typical example of programmer assumptions not being met by system deployment, leading to a resource access attack. Rule R3 restricts the entrypoint in `libdbus` to connect to only the trusted message bus labeled `system_dbusd_var_run_t` (in directory `/var/run/dbus`) for high-integrity processes, thus blocking the attack for all vulnerable `setuid` programs.

E4: PHP scripts. PHP local file inclusion (LFI) is a widespread attack caused by improper input filtering in PHP scripts, causing the PHP interpreter to load attacker-specified untrusted code. We setup Joomla!, a popular content management system written in PHP. A large number of third-party modules have been written for Joomla!, many improperly filtering input filenames, enabling the adversary to launch PHP LFI attacks (e.g., 82 CVEs in 2010 alone). Rule R4 restricts the instruction including files in the PHP interpreter to only open those of appropriate SELinux labels (`httpd_user_script_exec_t` by default on Ubuntu). We tested that an attack on the gCalendar component (Table 4) was blocked, but R4 should block all such attacks.

Openssh (E5) has a non-reentrant signal handler vulnerability that can be blocked using one set of system-wide rules (R9-R12). D-Bus (E6) and the Java compiler (E7) had unpatched vulnerabilities (E7 was known for at least two years but still unpatched). Rules R5, R6 in Table 5 block E6, and rule R7 blocks E7. The Process Firewall rules thus perform a function similar to dynamic binary patching [9].

6.1.2 New Vulnerabilities Found

E8: GNU icecat. One of the machines on which we installed the Process Firewall had the GNU Icecat browser. This had an insecure environment variable that caused it to search for libraries in the current working directory. The Process Firewall silently blocked this attack (rule R1); we noticed it later in our denial logs. We reported it to the maintainer who accepted our patch.

E9: init script. When examining accesses matching our `safe_open` rules (that we apply system-wide), we found one Ubuntu init script that unsafely created a file. The bug was accepted by Ubuntu and assigned a CVE.

6.2 Process Firewall Performance

We examine here the performance impact of the Process Firewall in two ways: (1) comparing the performance impact of blocking resource access attacks in the program vs. the Process Firewall and (2) the performance overhead of the Process Firewall relative to an unprotected system. First, we found that enforcing strong defenses against resource access attacks in the Process Firewall could be done much more efficiently in the Process Firewall than in programs. Second, we found that the Process Firewall including over 1000 rules incurs no more than a 4% overhead over a variety

Rule suggestions from runtime analysis

Only allow loading trusted library files by the dynamic linker.
R1: `pftables -p /lib/ld-2.15.so -i 0x596b -s SYSHIGH -d ~{lib_t|textrel_shlib_t|httpd_modules_t} -o FILE_OPEN -j DROP`
Load only trusted python modules.
R2: `pftables -p /usr/bin/python2.7 -i 0x34f05 -s SYSHIGH -d ~{lib_t|usr_t} -o FILE_OPEN -j DROP`
Allow the D-Bus library to connect only to trusted D-Bus server socket.
R3: `pftables -p /lib/libdbus-1.so.3 -i 0x39231 -s SYSHIGH -d ~{system_dbusd_var_run_t} -o UNIX_STREAM_SOCKET_CONNECT -j DROP`
Only include properly labeled PHP files (prevent local file inclusion attacks).
R4: `pftables -p /usr/bin/php5 -i 0x27ad2c -s SYSHIGH -d ~{httpd_user_script_exec_t} -o FILE_OPEN -j DROP`

Rule generation from known vulnerabilities

Protect D-Bus against a known vulnerability.
On bind, record the inode number created (using the STATE target module).
R5: `pftables -i 0x3c750 -p /bin/dbus-daemon -o SOCKET_BIND -j STATE --set --key 0xbeef --value C_INO`
On chmod, block if same inode is not used (using the STATE match module).
R6: `pftables -i 0x3c786 -p /bin/dbus-daemon -o SOCKET_SETATTR -m STATE --key 0xbeef --cmp C_INO --nequal -j DROP`
Disallow java from loading untrusted configuration files.
R7: `pftables -i 0x5d7e -p /usr/bin/java -d ~{SYSHIGH} -o FILE_OPEN -j DROP`

Manually-specified rules

Rule equivalent to SymLinksIfOwnerMatch Apache Configuration Option
On traversing symbolic links, check that link target owner is the same as the link owner of the link. Use COMPARE match module to compare.
C_TGT_OWNER is set by a context module that fetches details of symbolic link targets.
R8. `pftables -i 0x2d637 -p /usr/bin/apache2 -o LINK_READ -m COMPARE --v1 C_DAC_OWNER --v2 C_TGT_DAC_OWNER --nequal -j DROP`
Signal handler races
Main chain: Jump to signal chain if a signal is to be delivered to a process.
R9: `pftables -I input -o PROCESS_SIGNAL_DELIVERY -j SIGNAL_CHAIN`
If process is already executing a signal handler and signal to be delivered has a handler and is not unblockable (match module SIGNAL_MATCH), drop (possible race).
R10: `pftables -I signal_chain -m SIGNAL_MATCH -m STATE --key 'sig' --cmp 1 -j DROP`
Else, record that we are entering a signal handler in our state.
R11: `pftables -I signal_chain -m SIGNAL_MATCH -j STATE --set --key 'sig' --value 1`
When we return from a signal handler (sigreturn syscall), change state to indicate we are no longer in a handler.
R12: `pftables -I syscallbegin -m SYSCALL_ARGS --arg 0 --equal NR_sigreturn -j STATE --set --key 'sig' --value 0`

Attack-specific rule templates (Fields in angle brackets are filled to generate rules)

T1: **Restrict entrypoint to access only a set of resources**
Explicitly identify each entry point of low-integrity data
`pftables -I input -i <ept> -p <prog> -d ~<resource_set> -o <op> -j DROP`
T2: **Defend TOCTTOU race conditions**
Record resource accessed in check call
`pftables -I create/input -i <check_ept> -p <prog> -o <op> -j STATE --set --key <use_ept> --value C_INO`
On use call, if different resource accessed, drop
`pftables -i <use_ept> -b <binary> -o <op> -m STATE --key <use_ept> --cmp C_INO --nequal -j DROP`

Table 5: Process Firewall rules discussed in text (***R1 - R11***) and templates used to generate rules (***T1, T2***).

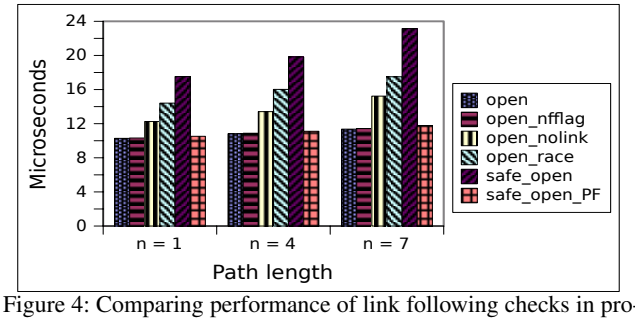

Figure 4: Comparing performance of link following checks in program or as Process Firewall rules as a function of path length n.

of macrobenchmarks and less than 11% overhead on any one system call.

System Call Performance. Figure 4 shows the performance of four variants of the `open` system call as a function of path length that provide differing protections against link following attacks. The average path length for our system was 2.3. The baseline `open` does not perform any checks. `open_nofollow` uses the O_NOFOLLOW flag, which prevents such attacks, but is non-portable and may also block desirable uses of symbolic links. `open_nolink` opens a file if it is not a link, by the sequence `lstat-open`. `open_race` eliminates the race between `lstat` and `open` by performing an additional `fstat` after opening the file. Lastly, `safe_open` performs such checks for each path component and is necessary to completely prevent link following attacks [8]. `safe_open_PF` is the equivalent of `safe_open` implemented using Process Firewall rules. While `safe_open` had overheads of up to 103% over the baseline `open` (for $n = 7$), our equivalent in the Process Firewall had a maximum overhead of only 2.3%. The overhead of `safe_open` is because it needs to perform at least 4 additional system calls for each path component. This shows how Process Firewall rules can powerfully extend the system call API with arbitrary resource constraints, while still maintaining performance. This also eliminates the need for checks in code, and thus, races.

Removing Program Checks. We use the Apache webserver to examine performance benefit in moving checks out of the program into the system at a macro-level on real-world programs. To protect against symbolic link following vulnerabilities, the option `SymLinksIfOwnerMatch` constraints Apache to follow a symbolic link only if both the

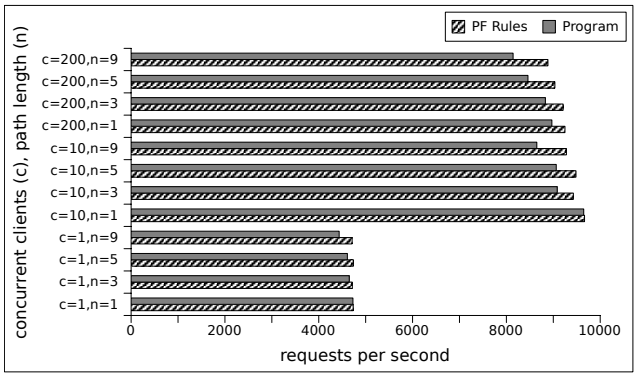

Figure 5: Comparing performance of Apache's SymLinksIfOwnerMatch option with Process Firewall rules as a function of path length (n) and number of clients (c). We measured requests per second for Apache 2.2.22 on a Dell Optiplex 845 serving a static web page averaged over 30 runs for each parameter combination.

symbolic link and the target file it refers to are owned by the same user. This reduces performance by forcing Apache to perform additional `lstat` system calls on each component of the pathname. Thus, the Apache documentation [2] recommends switching this option off for better performance. Furthermore, the documentation notes that this option can actually be circumvented through races.

Figure 5 compares the performance of these checks against an equivalent Process Firewall rule enforcing `SymLinksIfOwnerMatch` (R8 in Table 5). As both path length and number of concurrent clients increase, we note a performance improvement of Process Firewall rules over program checks – for a path length of 1 (`/index.html`), we noted a performance improvement of 3.02% for 200 concurrent clients; for path lengths 3, 5, and 9, it is 4.12%, 6.35% and 8.36%, respectively. The Process Firewall rule is thus both more efficient and secure.

Process Firewall Performance. We perform experiments on a Process Firewall implemented for Linux kernel 2.6.35 on a Dell Optiplex 980 with 2GB of RAM. Table 7 documents our macrobenchmarks. We measure both the Process Firewall with no rules (PF Base) and a rule base consisting of a set of 1218 rules (PF Full) generated by setting a lower threshold (100) for rule suggestion. Some benchmarks perform a large number of system calls (Apache Build and webserver benchmarks), while Bootup exercises a variety of rules in different ways. Each macrobenchmark shows between 2 and 4% overhead.

Table 6 shows overhead of the Process Firewall per system call. The microbenchmarks show no more than 0.51% overhead with just the default allow rules enabled (BASE) and less than 11% for any particular system call with our full rule set and optimizations enabled.

6.3 Rule Generation

This section examines the challenge of producing rules for the Process Firewall. The goal is generate rules automatically that block attacks without introducing false positives.

Benchmark	Mean ±95% CI (% overhead)				
	Without PF	PF Base		PF Full	
Apache Build (*s*)	73.67 ±0.06	72.82 ±0.12	(0.2)	75.61 ±0.06	**(4.0)**
Boot (*s*)	14.49 ±0.10	14.51 ±0.2	(0.0)	14.82 ±0.1	**(2.2)**
Web1-L (*ms*)	0.946±0.001	0.947±0.001	(0.1)	0.967±0.002	**(2.2)**
Web1-T (*Kb/s*)	467.67 ±0.1	465.45 ±0.3	(0.5)	455.35 ±0.8	**(2.5)**
Web1000-L (*ms*)	0.963±0.002	0.967±0.005	(0.4)	0.992±0.012	**(3.0)**
Web1000-T (*Kb/s*)	459.15 ±0.1	455.14 ±0.7	(0.9)	444.04 ±1.2	**(3.2)**

Table 7: Benchmark overhead means are collected over 30 runs. Web1-1000 indicates ApacheBench latency on a LAMP system serving random database entries with 1 and 1000 concurrent clients respectively (L is latency and T is throughput). PF Base has default allow rules, and PF Full uses our full rule set.

Since several systems (e.g., systrace [34], AppArmor [30], SELinux [1]) have used runtime analysis to automatically produce rules, we explore its effectiveness for the Process Firewall. In addition, we examine the ability and efficacy of OS distributors to produce effective rulesets automatically.

6.3.1 Rule Generation Techniques

We explore producing rules from known vulnerabilities, from runtime traces of program test suites, and from runtime traces of the system deployment. We find that: (1) known vulnerabilities can be blocked without incurring false positives; (2) for rules generated using three program test suites, we observed no false positives but these rules create unnecessary false negatives; and (3) while generating rules from runtime traces of program deployments reduces false negatives and generates widely-applicable rules, we observed that some false positives result. We examine causes for these false positives and future work to address them.

First, we generate rules for each of the over 20 previously-unknown vulnerabilities we found using our vulnerability testing tool [41]. Our testing tool logs the process entrypoint and the unsafe resource that led to the attack. We ran our testing tool to generate log entries for two verified exploits (E6, E7 in Table 4). We used the log entries to generate Process Firewall rules (R3, R4 for E7 in Table 5) to block the exploits and verified that they worked. The advantage of using known vulnerabilities to generate Process Firewall rules is that the combination of unsafe resource and entrypoint is known to require defense to protect the program, so no false positives are possible. We generalize the rules to deny access to all unsafe resources (see Table 2) for the program entrypoint based on the type of vulnerability, using the SELinux MAC policies of the program itself and system services (e.g., the untrusted search path rule R7 in Table 5 is generalized to block all adversary-accessible resources). These policies are essentially fixed, so as long as we have a conservative view of what is unsafe these rules do not cause false positives.

To generate rules to block unknown vulnerabilities, we explore rule generation from runtime traces using program *test suites*. Many programs include test suites written by developers to exercise their programs' functionality in a variety of ways. For example, PHP has a set of almost 8000 tests that exercise various configurations. We generated Process Firewall rules from runtime traces of the Apache, PHP, and MySQL test suites and did not observe false positives

Syscall	DISABLED	BASE		FULL		CONCACHE		LAZYCON		EPTSPC	
null	11.675	11.681	(0.05)	12.641	(8.27)	12.666	(8.48)	11.865	(1.62)	11.873	(**1.69**)
stat	12.545	12.609	(0.51)	26.403	(110.46)	23.207	(85.00)	21.981	(72.21)	13.872	(**10.57**)
read	11.767	11.805	(0.32)	15.332	(30.29)	14.823	(25.97)	13.704	(16.46)	11.982	(**1.82**)
write	11.763	11.794	(0.26)	13.602	(15.63)	13.096	(11.33)	12.123	(3.06)	11.975	(**1.80**)
fstat	11.826	11.134	(0.06)	15.529	(31.31)	14.897	(25.96)	13.719	(16.01)	12.106	(**2.36**)
open+close	24.632	24.722	(0.36)	44.54	(80.82)	39.789	(61.53)	37.511	(52.28)	26.113	(**6.01**)
fork+exit	104.326	104.392	(0.06)	112.371	(7.71)	110.918	(6.31)	106.303	(1.89)	104.959	(**0.60**)
fork+execve	664.010	667.050	(0.45)	903.714	(36.10)	861.571	(29.75)	818.142	(23.21)	670.589	(**0.99**)
fork+sh -c	1461.875	1465.375	(0.23)	1934.666	(32.34)	1856.120	(26.96)	1766.320	(20.80)	1475.475	(**0.89**)

Table 6: Microbenchmarks using lmbench. All results are in μs with percentage overhead in brackets. Standard deviation for all results was less than 1% for all our measurements. 95% confidence intervals for the non-fork results was at maximum 0.003, whereas for the fork-related results, the maximum was 0.3. Each column except the last incorporates optimizations of the previous column. DISABLED is the Process Firewall totally disabled, BASE with only the default allow rule, FULL our full rule base without any optimizations, CONCACHE with context caching optimization, LAZYCON with lazy context evaluation, and EPTSPC with entrypoint-specific rule chains.

Invocation Threshold	High Only	Low Only	Both High and Low	Rules Produced	False Positives
0	4570	664	0	5234	525
5	4436	508	290	2329	235
10	4384	482	368	1536	157
50	4257	480	497	490	28
100	4247	480	507	295	18
500	4233	480	521	64	4
1000	4230	480	524	34	1
1149	**4229**	**480**	**525**	**30**	**0**
5000	4229	480	525	11	0

Table 8: Classification of entrypoints against the number entrypoint invocations (one invocation is one system call).

when enforcing these rule sets. However, test suites exercise programs under multiple program *environments* – configurations, command line arguments, and environment variables [22]. These environments may access resources that are not relevant to the expected deployment, thus resulting in rules that cause false negatives. For example, the Apache test suite exercises programs under configurations that allow and disallow low-integrity user-defined configuration files (`.htaccess`). If the expected deployment disallows `.htaccess`, this rule may miss attacks where Apache is somehow tricked into using these low-integrity files. Test suites have other drawbacks – their quality is variable, and they are not available for all programs.

To reduce the number of false negatives, we examine rule generation using runtime traces from deployed program environments. However, using runtime traces may cause false positives as a trace may not exercise all valid resource accesses by an entrypoint. We analyzed how entrypoints accessed resources over a two-week long runtime trace on an Ubuntu 10.04 system with SELinux that had 5234 total entrypoints and 410,000 log entries. From Section 4.1, invariant rules to prevent several resource access attacks can be generated for those entrypoints that access only either high-integrity (adversary-accessible) or low-integrity (adversary-inaccessible) resources, but not both. To generate such rules, we collected all resources accessed by each entrypoint in the runtime trace. Depending on whether these resources were only high-integrity, low-integrity, or both, the entrypoint was classified as high, low, or both. From this classification, rules were generated for entrypoints that were: (1) classified as either high or low, and (2) invoked more than a threshold number of times. For this study, we defined any resource modifiable by processes running under the untrusted SELinux user label `user_t` as low-integrity.

We now examine whether the runtime trace identifies a threshold beyond which no false positives are observed. False positives are caused when entrypoints are classified as either high or low, but in reality access both. Table 8 shows how the classification of entrypoints evolved with the number of invocations of that entrypoint in the trace. The highest number of invocations at which an entrypoint changed class from high or low to both was 1149. Thus, if we used 1149 as the threshold for producing rules, then we would not see any false positives for this particular runtime trace. While only 30 entrypoints are invoked 1149 times (or more) and are classified either high or low in this trace, the corresponding rules apply in many cases (e.g., dynamic linking, PHP File Inclusion, etc.). Rules (R1-R4) in Table 5 were all generated based on entrypoints that were invoked more than 1149 times.

If rules are generated using a lower threshold, many more entrypoints could be protected. To find causes for false positives at lower thresholds, we manually examined the 28 entrypoints that changed classification after more than 50 invocations. First, 18 entrypoints were in libraries. These occur because libraries are called by a variety of programs in different environments, which may use the libraries for different purposes. Thus, these rules must be predicated on the environment in which the library is used. The remaining 10 were program entrypoints. Although these programs were launched under the same environment every time, they used inputs at runtime to produce *names* to access resources. For example, an entrypoint in `nautilus`, a graphical file browser, lists the files in a directory the user specifies in the location bar. In our particular runtime trace, the user only accessed high-integrity files in the first 50 invocations, and later accessed a low-integrity directory. Thus, to guarantee generation of rules without false positives, we need to understand how a program produces names used in resource access system calls. We leave this for future work.

6.3.2 Rule Generation by OS Distributors

We envision that OS distributors will generate Process Firewall rules and ship them to users in application packages. In this section, we discuss how to automate rule generation and whether the techniques above are useful for effective rule generation by OS distributors.

We provide scripts to automatically generate rules from Process Firewall logs (generated by the `LOG` target module). Table 5 shows two rule templates (T1, T2) we use for rule generation from known vulnerabilities. Template T1 constrains an entrypoint to access only a set of resources identified by their (SELinux) labels[5]. T2 creates rules to block TOCTTOU attacks. Scripts fill in rule template fields using corresponding logged values. To generate rules from known vulnerabilities using these rule templates, we need the specific Process Firewall log entries for the vulnerable system call and the type of the vulnerability. The type of vulnerability determines the template, which we fill with logged data.

An important question is whether rules generated by OS distributors are valid in deployed environments. Our insight is that Process Firewall rules generated by OS distributors are valid if programs are run in the same environment that the OS distributors generate rules for. To find such programs, we compared the launch inputs and application package files across all program invocations. If every launch used the same command line arguments and environment variables and the package files were unmodified from installation[6], we concluded that the deployed environment was consistent with the OS distribution's package. Of the 318 programs and scripts that were launched in our runtime trace, we found that 232 were launched in the same environment as the installed package each time. Based on our analysis above, OS distributors could produce Process Firewall rules without false positives for a majority of programs, where they would only need to resolve 10 false positives (e.g., by testing the programs more thoroughly). Alternatively, OS distributors could produce less comprehensive rule sets using Linux test suites or only block known vulnerabilities with little risk of false positives.

7. Related Work

While no previous work has looked at the class of resource access attacks in a unified manner, there exist previous system and program defenses.

System-level defenses. System-level defenses have been proposed for confinement and protection. Confinement using sandboxing [16, 21, 23, 34] cannot trust process context to make authorization decisions, as malicious processes could spoof such information. Some systems [37, 38] assume "partially trusted" processes and use the process call stack. However, they do not make use of system information such as adversary accessibility in addition; this precludes them from defending against resource access attacks. System-level defenses have also been proposed for protection against TOCTTOU attacks [5, 11, 17, 39, 44] and link following attacks [8, 44]. However, system-level protection is fundamentally limited because it does not consider program context [7].

Program Defenses. Certain program API modifications have been proposed to help programs convey constraints on resource access to the OS depending on process context. Decentralized Information Flow Control [26, 46] (DIFC) enables programmers to limit the system resources available in different process execution contexts through information flow policies. Capability systems [27, 43] circumvent resource access through namespaces altogether by using direct capabilities to resources. However, such defenses require programs to be customized and rewritten to system deployments. The Process Firewall performs this customization without requiring program modifications.

Programs restrict resource access attacks such as untrusted search paths, directory traversal and PHP file inclusion by filtering adversarial names (e.g., removing `../`). However, both locating all sources of adversarial input [40] and proper input filtering of names [4] are hard problems, in addition to being deployment-specific. The Process Firewall solves these problems by using resource information to block access, instead of trying to restrict names.

8. Conclusion

This paper introduced the Process Firewall, a system-wide kernel protection mechanism that protects processes from a variety of resource access attacks. Noting the complexity, inefficiency and incompleteness of current program defenses, our insight is to use both system knowledge of resources and adversary access, and program context to *protect* processes from resource access attacks. To this end, we implemented a Process Firewall prototype for the Linux kernel, utilizing a variety of optimizations that resulted in overheads of less than 4% *system-wide* for a variety of macrobenchmarks. In addition, we found that it is more efficient to deploy resource access defenses in the Process Firewall than in programs. Finally, we show that the Process Firewall is easy to use, as no program changes or user configuration are required and rule bases can be created from existing vulnerabilities and runtime analysis to avoid false positives. These results show that it is practical for the operating system to protect processes by preventing a variety of resource access attacks system-wide.

Acknowledgments

We thank Nigel Edwards, Mathias Payer, our anonymous reviewers, and in particular, our shepherd Petros Maniatis, for their comments that helped significantly improve the presentation of the paper. We acknowledge support from grants NSF-CNS-0905343 and AFOSR-FA9550-12-1-0166.

[5] We could also have used the DAC label (owner and group) to identify resources, but we chose SELinux as its labels are finer-grained

[6] User-defined configuration files are also a sign that the environment may differ across runs.

References

[1] audit2allow. http://fedoraproject.org/wiki/SELinux/audit2allow.

[2] Apache Performance Tuning. http://httpd.apache.org/docs/2.2/misc/perf-tuning.html#symlinks, 2012.

[3] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, AFSC, October 1972.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE SSP*, 2008.

[5] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[6] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security '06*, 2005.

[7] X. Cai, Y. Gui, and R. Johnson. Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE SSP*, 2009.

[8] S. Chari, S. Halevi, and W. Venema. Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *NDSS '10*, 2010.

[9] H. Chen, J. Yu, R. Chen, B. Zang, and P. chung Yew. POLUS: A POwerful live updating system. In *29th Intl Conf. on Software Engineering*, pages 271–281, 2007.

[10] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman, 1994.

[11] C. Cowan, S. Beattie, C. Wright, and G. Kroah-hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *USENIX SSYM*, 2001.

[12] Cryogenic-Sleep. Symlinks and Cryogenic Sleep. http://seclists.org/bugtraq/2000/Jan/16, 2000.

[13] CVE. Common vulnerabilities and exposures. http://cve.mitre.org/, 2012.

[14] CWE. Common weakness enumeration. http://cwe.mitre.org/, 2012.

[15] CWE-22. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'). http://cwe.mitre.org/data/definitions/22.html, 2012.

[16] David S. Peterson and Matt Bishop and Raju P. A flexible containment mechanism for executing untrusted code. In *USENIX Security*, 2002.

[17] D. Dean and A. Hu. Fixing races for fun and profit. In *USENIX Security*, 2004.

[18] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE SSP '04*, 2004.

[19] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE SSP '96*, 1996.

[20] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *NDSS*, 2003.

[21] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[22] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *RAID '06*, 2006.

[23] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security '96*, 1996.

[24] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *USENIX Security*, 2003.

[25] T. Jaeger, A. Edwards, and X. Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Trans. Inf. Syst. Secur.*, 2004.

[26] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Frans, K. Eddie, and K. R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[27] H. M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[28] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Syst. J.*, 1974.

[29] B. Niu and G. Tan. Enforcing user-space privilege separation with declarative architectures. In *ACM STC*, 2012.

[30] Novell. AppArmor Linux Application Security. http://www.novell.com/linux/security/apparmor/.

[31] NSA. SELinux, 2012. http://www.nsa.gov/selinux.

[32] J. Olsa. LKML: kernel: backtrace unwind support. https://lkml.org/lkml/2012/2/10/129, 2012.

[33] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE S&P*, 2012.

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

[35] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security*, 2003.

[36] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[37] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE SS&P '01*, 2001.

[38] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP '03*, 2003.

[39] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX FAST*, 2008.

[40] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. In *ASIACCS*, 2012.

[41] H. Vijayakumar, J. Schiffman, and T. Jaeger. STING: Finding Name Resolution Vulnerabilities in Programs. In *USENIX Security*, 2012.

[42] D. Wagner. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM CCS*, 2002.

[43] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: practical capabilities for UNIX. In *USENIX Security*, 2010.

[44] J. Wei and C. Pu. Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems. *Computers & Security*, 2010.

[45] C. Wright, C. Cowan, and J. Morris. Linux security modules: General security support for the Linux kernel. In *USENIX Security*, 2002.

[46] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

[47] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security*, 2002.