# BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System

SURESH N. CHARI and PAU-CHEN CHENG
IBM Thomas J. Watson Research Center

Detecting attacks against systems has, in practice, largely been delegated to sensors, such as network intrusion detection systems. However, due to the inherent limitations of these systems and the increasing use of encryption in communication, intrusion detection and prevention have once again moved back to the host systems themselves. In this paper, we describe our experiences with building BlueBox, a host-based intrusion detection system. Our approach, based on the technique of system call introspection, can be viewed as creating an infrastructure for defining and enforcing very fine-grained process capabilities in the kernel. These capabilities are specified as a set of rules (policies) for regulating access to system resources on a per executable basis. The language for expressing the rules is intuitive and sufficiently expressive to effectively capture security boundaries.

We have prototyped our approach on Linux operating system kernel and have built rule templates for popular daemons such as Apache and wu-ftpd. Our design has been validated by testing against a comprehensive database of known attacks. Our system has been designed to minimize the kernel changes and performance impact and thus can be ported easily to new kernels. We describe the motivation and rationale behind BlueBox, its design, implementation on Linux, and how it relates to prior work on detecting and preventing intrusions on host systems.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; *Information flow controls*; *Invasive software*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software*; *Unauthorized access*

General Terms: Security, Design, Experimentation, Performance

Additional Key Words and Phrases: Intrusion detection, sandboxing, policy, system call introspection

## 1. INTRODUCTION

The two mechanisms predominantly used to secure application servers today are firewalls and network intrusion detection systems. One of the attractive features of these mechanisms is that they are independent of the server, and thus easily deployed. Firewalls control the flow of communication to systems and network IDSs detect possible attacks by monitoring this communication. While properly configured firewalls serve their intended purpose, current network

A preliminary version of this paper appeared in NDSS'02 [Chari and Cheng 2002].
Authors' address : S. N. Chari and P.-C. Cheng, IBM, P.O. Box 704, Yorktown Heights, NY 10598; email: schari@us.ibm.com, pau@watson.ibm.com.
Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appearance, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2003 ACM 1094-9224/03/0500-0173 $5.00

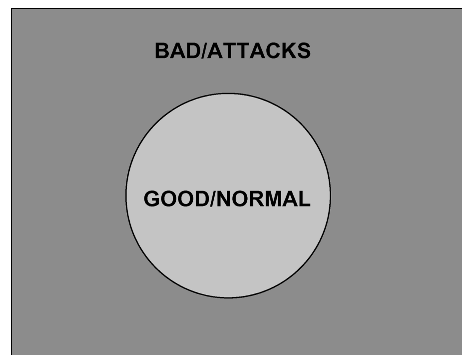


Fig. 1. Ideal IDSs.

IDSs suffer from a number of limitations. Network IDSs typically analyze traffic on the network either by scanning for patterns containing known attacks or detecting statistically abnormal traffic patterns. With the widespread use of traffic encryption protocols, such as SSL [Freier et al. 1996; Dierks and Allen 1997] and IPSEC [Atkinson 1995], a significant portion of traffic on the Internet is encrypted and therefore is unavailable for examination in the clear. Also, there are well-known ways to evade network IDSs [Ptacek and Newsham 1998]. Thus, increasingly, intrusion detection must move to the host server where the content is visible in the clear and these evasion techniques do not work. Our system, BlueBox, is such a host-based real-time intrusion detection system and it can also be configured for blocking intrusions.

In order to contrast with the mechanisms used in our system, we first look at mechanisms used in currently deployed host-based IDSs. They are primarily based on one of the following [Debar et al. 1999; Jackson 1999]:

*Anomaly Detection:* Such systems first try to characterize a statistical profile of "normal" behavior [Javitz and Valdes 1994; Anderson et al. 1993; Forrest et al. 1996; Debar et al. 1998]. A pattern that deviates significantly from the normal profile is considered an attack.

*Misuse Detection:* These systems first defined a collections of signatures ( representative patterns) of known attacks [Jackson 1999; Paxson 1998; Ranum et al. 1997; Crosbie et al. 1996]. Activities matching such patterns are considered attacks.

Figures 1 and 2 depict the two approaches in an ideal world and in reality. Conceptually, misuse detection is based on knowledge of bad behavior (attacks) and anomaly detection is based on knowledge of good (normal) behavior. If both techniques were perfect, then each would exactly complement the other: that is, what is not bad is good and vice versa. In reality, neither technique is perfect. Misuse detection can never know all possible attacks and it usually classifies some good behavior as attacks. Likewise, anomaly detection cannot cover all good behavior and will mistake some attacks for good behavior. Also, an entity's behavior profile will change as its usage pattern changes. So anomaly detection has to adapt its profile to these changes. This opens the possibility

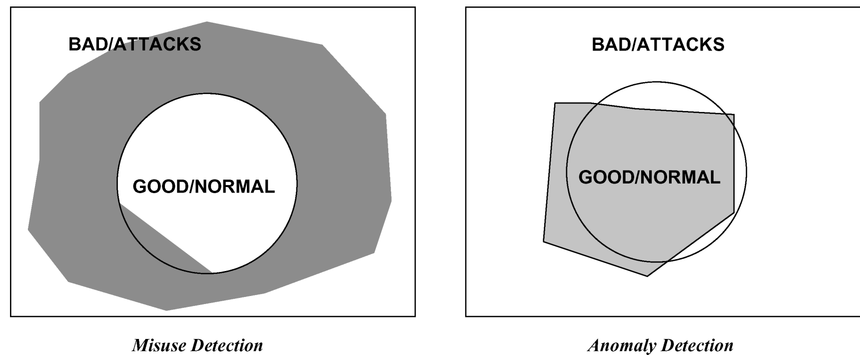


Fig. 2. IDS's in reality.

for an attacker to gradually increase its level of malicious activities until these activities are considered normal.

Our policy-driven technique, similar to the concept *sandboxing*, tries to define the boundary between the good and the bad as a set of rules. These rules specify what an executable program or script is allowed to do and attempts to violate them are considered intrusions. The rules governing a process define precisely which system resources a process can access and in what way. Section 2 gives an overview of what the scope of the rules are. The rules are defined through precise understanding of the expected behavior of the program. They can be defined using existing templates, audit trails, configuration, and, if necessary, program semantics. The rules are specified off-line, compiled into a machine readable binary that is associated with the program and loaded into the kernel when the program is executed. Rule enforcement happens when the program is executed in the context of a process: the behavior of the process is checked and constrained according to the rules. The enforcement is done in the kernel during invocations of system calls. The concept of sandboxing has appeared in numerous contexts including IDS and we discuss this in Section 3. In particular, systems such as LIDS [Xie and Biondi 2001], SubDomains [Cowan et al. 2000] and REMUS [Bernaschi et al. 2000, 2002] follow similar principles. In Section 3 we will describe these and other approaches such as Linux Security Module [LSM ] and the Secure Linux Project [SELINUX ] and highlight how these systems differ from BlueBox.

We believe that the policy-based approach of BlueBox and like systems offers a number of advantages over the traditional attack–signature based or profile-based approaches. They include:

—The security boundary is much more precisely defined in terms of the intended use of the sensitive system resources. Rules are based on understanding a program's behavior and *not* on attack signatures or time-variant, incomplete statistical profiles of "normal" behavior. This offers two benefits
  1. *Unknown attacks* can be detected.
  2. Previously unseen but legitimate behavior would not be mistaken for attacks.

Therefore the false positive and, potentially, false negative rates will be lower.

—Another potential win is the manageability of the IDS, especially as compared to statistical profiling based techniques. There is no need to constantly maintain and update attack–signature database or statistic profiles. Since rules are precisely defined in terms of system resources, there will be very few updates, if any, of rules for an application running on a particular platform.

—Perhaps the most important advantage of BlueBox's policy-based approach is that detection is done in *real time*. Therefore the system could, optionally, block an unauthorized access or action.

On the other hand, since the rules for an application are defined by its access to system resources there are some disadvantages when compared with other IDSs. These include:

—*Version Migration*: Since different versions of applications may access different resources every version will require modified sets of rules. However, in our experience with the Apache http server, minor version changes impact the rules very minimally. Even with major version changes, large chunks of the rule sets can be reused.

—*In Memory Attacks*: Since the checks on process behavior are made only when the process makes a system call, attacks which are "in memory" cannot be detected.

The rest of the paper is organized as follows: Section 2 gives an overview of the specification and generation of rules in BlueBox. Section 3 surveys related work and compares a number of similar systems with BlueBox. Section 4 presents the technical details of our design and implementation, the precise scope of rules, and the system architecture. Section 5 presents a few examples of how BlueBox thwarts several well-known attacks and also provides detail experiences on specifying rules. Section 6 informally argues the soundness and power of BlueBox rule-checking. Section 7 discusses the performance impact of the IDS. Section 8 discusses future research directions, and we conclude in Section 9.

## 2. BLUEBOX POLICY SPECIFICATION AND GENERATION

Since an attack on a system must access sensitive system resources in unintended ways to be successful, a BlueBox policy defines and enforces rules controlling a process's access to system resources, thus thwarting unintended access. We categorize system resources and the types of access to them in Table I.

Features of our current rule specification includes:

—Access permissions to file system objects.

—Access to the file system, for example, mount and unmount.

—Permitted *uid* and *gid* transitions.

—signals which can be sent, received, blocked, ignored, and handled.

Table I. Types of Resources and Access

| Resources | Types of Access |
|---|---|
| File system objects | create, open, read, write, *execute*, *removal*, *link-to*, *change of access permissions*, *change of ownership* |
| File systems | mount, unmount, types of mount |
| Identities | acquire, release, inherit |
| Processes (address spaces, signals, ...) | read, write, deliver |
| CPU cycles, process scheduling priority | raise |
| System clock | set, read |
| System/kernel memory | read, write |
| IPC objects: pipes, semaphores, message queues, shared memory, ... | create, open (attach), read, write |
| Devices, network | create/attach, open, read, write, *io-control*, *removal*, *link-to*, *change of access permissions*, *change of ownership* |
| Privileges | acquire, release, raise, lower |

—Process characteristics such as scheduling priorities that can be modified.

—Elementary controls for other system resources such as IPC objects, sockets, and ioctl calls. This is an area that requires more study for more comprehensive rules.

To make the policy specification more expressive, we provide an *allowed system calls list* as a coarser level of control that is effective in thwarting a number of attacks. Since system resources must be accessed through system calls, disallowing invocations of a system call disallows access to resources. For instance, most server processes do not need to mount or unmount file systems, so mount and unmount are not in their allowed lists, and an invocation of either will be considered an intrusion regardless of the invocation's parameters. Even this coarse form of access control is sufficient to effectively control the behavior of simple programs, such as cgi-scripts which execute on a web-server since they are typically very simple and offer limited functionality thereby making very few system calls.

In the list of system calls allowed to a process, a significant fraction have no exposure in that they are mainly used to query for information. While the information queried in these calls can be considered sensitive in some situations, we note that correct rules on other system calls will guarantee that this information will only be used in a controlled manner. For our implementation on the Linux kernel, we have identified 72 *harmless* system calls: each of which either has no security implications or is not supported by the Linux 2.2.14 kernel. These calls are listed in Table II.

The policy for a program can optionally be marked *inheritable*. This is very useful in several situations. Consider a cgi-script executing on a web-server: most scripts typically utilize a number of general programs such as ls and cat. Clearly, we cannot attach access control rules to these general programs. Marking the rules attached to the script as inheritable allows us to enforce the same policy during the invocations of these general programs. Another use of inheritable rules is when we want to share the same rules

Table II. *Harmless* System Calls

| | | | |
|---|---|---|---|
| afs_syscall | getgid | lstat64 | sched_yield |
| alarm | getgroups | mpx | setitimer |
| break | getitimer | msync | sgetmask |
| brk | getpgid | nanosleep | stat |
| capget | getpgrp | newselect | stat64 |
| chdir | getpid | oldfstat | statfs |
| fchdir | getppid | oldlstat | stty |
| fdatasync | getpriority | oldolduname | sysfs |
| fstat | getresgid | olduname | sysinfo |
| fstat64 | getresuid | poll | syslog |
| fstatfs | getrlimit | prof | time |
| fsync | getrusage | query_module | times |
| ftime | getsid | readlink | uname |
| get_kernel_syms | gettimeofday | sched_get_priority_max | ustat |
| getcwd | getuid | sched_get_priority_min | vfork |
| getdents | gtty | sched_getparam | vhangup |
| getegid | lock | sched_getscheduler | vm86 |
| geteuid | lstat | sched_rr_get_interval | wait4 |

across a number of programs. Inheriting rules eliminates duplication and is also more efficiently implemented since we do not parse the binary rules each time.

Based on our experience, for a given program, there are several mechanisms and tools one could use to build and specify the rules.

—*Intended Semantics*: The most comprehensive way to generate the correct rules for a program is by looking at the intended semantics of the program. While this can be daunting for big servers such as Apache, we have found that for several cgi-bin scripts, this is the easiest way to capture rules since these scripts typically access few resources.

—*Configuration*: For servers, such as Apache, that can be configured to run in different ways, configuration files need to be used (either manually or automated) to create rules.

—*Audit Trails*: A very straightforward mechanism to generate large chunks of the rules is to inspect system call audit trails. For a number of servers and scripts we have found this to be the simplest method.

—*Existing Templates*: For large and popular servers, such as the Apache httpd, we envision existing rule templates that can automatically be customized to new installations. Our reference server is the Apache httpd for which we have developed a template. We are currently investigating rule templates for larger application servers and hope to include rule templates for the most common configurations of application servers such as the IBM WebSphere [WEBSPHERE 2001].

While these mechanisms sound daunting for nontrivial programs, as we discuss in Section 5, we believe that the amount of extra work is manageable. For our prototypical application of web servers, most of the rules need to be done once, with little customization for new servers.

## 3. RELATED WORK

Restricting program behavior based on externally specified rules has a very long history dating back to the reference monitors of operating systems several decades ago. In this section, we highlight more recent mechanisms and compare them with our work. Some of the systems are very different from BlueBox while others are very similar.

### 3.1 Language-Based Mechanisms

There are a large number of language-based mechanisms to restrict program behavior based on policy. They range from the theoretical program correctness methodology of using asserts to the popular type-based mechanisms enforced by the loader such as the famed Java Virtual Machine [JVM 2001]. While the security guarantees promised by these mechanisms are stronger than ours, they make very strong and in some cases, unrealistic, assumptions about the trusted computing base (TCB). Some classes of such systems include the following:

3.1.1 *Program Correctness-Based Mechanisms.* This method has been the subject of extensive research spanning decades. Recently, these mechanisms have been proposed as effective mechanisms to mitigate exposures [Erlingsson and Schneider 2000]. While theoretically elegant, they are largely restricted to checks in the user space. Hence, the TCB needed for these mechanisms to be effective is unrealistic since all the checks inserted into the user space program *must* be executed. This is rarely realized in commercial operating systems: an attacker mounting a buffer overflow attack is in no way restricted by any of the checks inserted in the original program.

3.1.2 *Type-Based Mechanisms.* The celebrated Java Virtual Machine is a classic example of a system that enforces strong checks on the interpreted byte code. For this mechanism to work one has to extend the TCB to include the interpreter and loader. In several controlled environments this is possible, however, it is not realistic, for reasons of performance, to have daemons such as the http server run in this environment.

### 3.2 System Call Pattern-Based Systems

These systems identify intrusions by an initial training phase where exhaustive testing is used to identify the accepted set of patterns in system call sequences, and then flagging an intrusion if there are erroneous patterns in system calls made by daemons in an actual run. Some examples are discussed in Forrest et al. [1996] and Debar et al. [1998]. The main advantage of these systems is the minimized impact on the kernel, that is, one needs to make few changes to the kernel to implement them. However, they cannot offer strong security guarantees: firstly, their efficacy requires exhaustive training to identify normal patterns and if not done correctly, can result in a large number of false positives. Secondly, they are very sensitive to the exact version of the monitored software: small changes in source code can yield very different system call patterns. For example, the Apache http daemon can be configured to run using processes or

threads, and the system call patterns are considerably different. Since BlueBox tries to capture the resources the daemon uses, there are very few changes between the two versions.

### 3.3 Kernel-Based Reference Monitors

In the last few years there has been a renewed interest in sandboxing by intercepting system calls made by processes. We describe some systems and highlight the similarities and differences.

3.3.1 *LIDS.* The Linux Intrusion Detection system (LIDS) [Xie and Biondi 2001] aims to extend the concept of capabilities present in the basic Linux system by defining fine-grained file access capabilities for each process. BlueBox's rules for file system objects is very similar to LIDS. The complete rule set of BlueBox is a strict superset of the LIDS system. Among the several additional features of BlueBox is the state information, which is useful in thwarting some attacks as described in Section 5.

3.3.2 *LSM and SeLinux.* The Linux Security Module (LSM) [LSM] is an ambitious project to define an infrastructure to implement very fine-grained access control within the Linux kernel. LSM identifies a number of points within the kernel code where resources are accessed by the calling process and defines relevant callouts from these points to a pluggable module which actually implements policy by performing checks on these accesses. SeLinux [SELINUX] is one module that implements a mandatory access control policy using the infrastructure of LSM.

Since LSM defines checks interspersed with kernel code, implementing access control policies such as SeLinux on top of LSM can yield very strong security properties. For instance, since the checks are made just prior to the actual time of access, it avoids a potential security hole in BlueBox like approaches. This issue is discussed in detail in Section. 6. However, the impact of the LSM like approach on kernel code is much greater than that of BlueBox. One of the key design principles of our system is to minimize, as far as possible, the impact on the kernel.

3.3.3 *A Program as a Finite State Machine.* Sekar and Uppuluri [1999] present a system which combines language-based systems with system call intercept-based systems. Their approach is to model processes with a state diagram describing its functionality and then enforcing this state diagram in the kernel during system call invocation. They achieve strong security guarantees since the state diagram captures exact process semantics. The main drawback of this system is the difficulty in generating the required state diagrams for a new process. Also, we conjecture, based on our experience in incorporating state, that the performance penalty in enforcing the rules could be somewhat high.

3.3.4 *Generic Software Wrappers.* Generic software wrappers [Ko et al. 2000; Fraser et al. 1999] are a mechanism to enforce various access control and intrusion detection checks triggered by events during process execution. The

infrastructure will register various scripts to be run based on events, monitor process execution for these events to occur, and execute registered scripts when the events occur. This is a powerful infrastructure that can integrate numerous approaches to system security under one unifying framework. The main drawbacks of this approach is the complexity of writing scripts and the performance impact in such a complex framework. We believe that our approach is much more intuitive and has substantially better performance.

3.3.5 *SubDomains.* Cowan et al. [2000] define the concept of SubDomain: a kernel-enforced refinement of the underlying operating system checks on file system access. Like BlueBox, this set of refinements is defined and enforced on a per-program basis as opposed to a per-user basis. Roughly, the expressive power of the SubDomain rules correspond to the file system object rules of BlueBox. Thus, the rules controlling the behavior of other system calls and state maintenance mechanisms of BlueBox make it more expressive than the SubDomain model. However, making the enforcement simpler gives the SubDomain model better performance. The functional placement of the enforcer and other infrastructure in the kernel is also very different from that in BlueBox.

3.3.6 *Other Sandboxing Systems.* The system that comes closest to our system is REMUS [Bernaschi et al. 2000, 2002]. Its system architecture is very similar to ours and the main differences are in the syntax and semantics of the rules themselves. The placements of different parts of the system within the kernel are also very different. Our placement aims to minimize impact on the kernel code by placing a *wrapper* around kernel system call handlers while their placement tries to minimize performance impact. Our system is extensible to newer versions of the kernel since by and large the same wrapper should work for newer kernels.

The Domain-and-Type-Enforcement (DTE)-based system of Walker et al. [1996] groups file system objects into sets called *types* and puts a subject (an executable) into a domain which has specific access rights to types. It does not provide protection on nonfile-system-object resources and seems to incur more complexity when providing fine-granularity control than BlueBox.

### 3.4 User Space System Call Introspection

A valid criticism of systems such as BlueBox is the modifications to the kernel required to install the infrastructure to install and enforce process behavior rules. To circumvent this, one approach is to use existing monitoring infrastructure in kernels such as ptrace to have monitors which reside in user-space [Jain and Sekar 2000; Wagner 1999]. The monitor sits in a separate process and intercepts system calls made by the monitored process using ptrace; the monitor process can then enforce the rules by examining the intercepted system calls and their parameters and possibly modifying the parameters or terminating the calls. As pointed out by the authors [Jain and Sekar 2000], this approach has a few drawbacks. Firstly, since rules are enforced in the context of the monitor process, there is some overhead due to context switching and copying

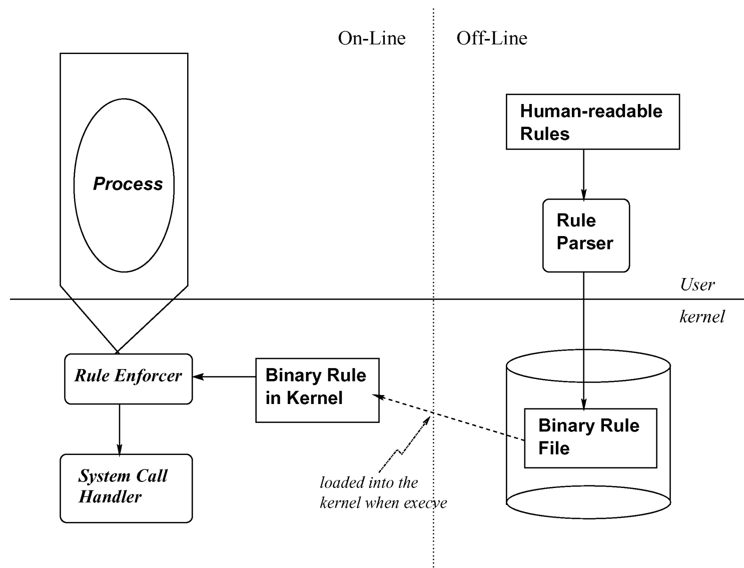


Fig. 3. BlueBox architecture.

data from one process's context to the other's. Also, there are cases when the monitored process is not entirely under the control of the monitor due to the implementations of ptrace.

## 4. TECHNICAL DETAILS

In this section, we will first discuss the BlueBox system architecture to show how a policy is defined and enforced, then we discuss policy specification in details, and conclude with a discussion of BlueBox's impact on the kernel.

### 4.1 System Architecture

The BlueBox system architecture is shown in Figure 3. The architecture includes two parts :

*Policy Specification and Parsing.* A BlueBox policy for an executable program is specified in a human-readable form using a text editor and then parsed into a binary file by a parser program. This part is done off-line and before the program is executed. Details are in Section 4.2.

*Policy Loading and Enforcement.* Since BlueBox policies are meant to control access to system resources that can only be accessed through system calls, the natural place for rule enforcement is at the kernel system call entry point. Our first prototype on Linux 2.2.14 places an *enforcer* module at the kernel system call entry point to enforce rules. We have since upgraded to Linux 2.4.18 and moved the entire enforcer into loadable kernel modules and thus avoided any change to the existing kernel code; Appendix gives more details on the kernel modules. The enforcer has built-in knowledge of what categories of resources each call may access, so it can check the parameters of the invoked

system call against the rules. It is our design principle to make the enforcer a *wrapper* around the kernel system call handlers rather than code insertions inside the handlers. Besides requiring no change to the existing handlers' code,[1] this principle focuses on the very stable syntax and semantics of the system call interface and not on the handlers' code which evolves much faster. Therefore the wrapper can be easily ported to newer versions of kernels. This is in direct contrast with the design philosophy of access control infrastructures such as the Linux Security Module [LSM].

Since it is impractical to write policies for all processes on a system, we added a new system call to mark a process as being *monitored*; this status will be passed on to its children and cannot be unmarked. As a tool, we have a simple wrapper program which marks itself as monitored and then execves the real program to pass on the *monitored* status to the new process image. When loading the new image the modified execve system call handler[2] loads the rules into the kernel and starts enforcement. If no rules for the new image are found, then the process will try to inherit and share the rules of the old image; if these rules are not inheritable or do not exist, then the process will be *crippled*; that is, it is only allowed to make harmless system calls.

Rules are *read-only* after being loaded. Each monitored process is allocated a kernel memory buffer[3] to hold its private BlueBox state that can change as the process executes. More discussion on BlueBox process state is given in Section 4.3. When a process forks, the child process shares the parent's policy but will be given a copy of the parent's BlueBox state. A process's BlueBox state will be reset when it execves a program.

### 4.2 Rules for Different Types of Resources

In this section, we will discuss rules for various classes of system resources such as file system objects, uid/gid lists, signals, sockets, and device special files. Each class has its own particular syntax and semantics. The description of the rule syntax for these resources and their associated semantics captures the essence of almost all of the possible rules under BlueBox.

4.2.1 *Rules for File System Objects.* Rules on file system objects are encoded as a tree which mimics the hierarchy of files on a UNIX system. The policy of a program includes one such tree encoding the program's access rights to file system objects. Figure 4 shows a part of the specification of rules on file system objects for Apache 2.0 HTTP server. Each node in the tree records access rights to a (set of) file system object(s). The root of a tree corresponds to the root of the hierarchy of files. Like a UNIX file system, each node has a name. Unlike a UNIX file system, the name can contain UNIX shell-like *wildcard* characters "*" and "?" with the same interpretation as in a UNIX shell. The only exception is that a leaf node with the name "∗" represents an entire subtree; for

[1] The handlers of `execve`, `fork`, and `exit` are changed to load, share, and unload rules, respectively.
[2] The API for execve is not changed.
[3] At present, the size of the buffer is one page or 4 KB.

| pathname | access permisions | creation mode |
|---|---|---|
| /* *r:read, w:write, x:execute, c:create, a:append* */ | | |
| /* share libraries */ | | |
| /etc/ld.so.* | *r* | |
| /lib/* | *r* | |
| /* system configuration files */ | | |
| /etc/host.conf | *r* | |
| /etc/hosts | *r* | |
| /etc/passwd | *r* | |
| /etc/group | *r* | |
| /etc/resolv.conf | *r* | |
| /* Apache files */ | | |
| /usr/local/apache2/conf/* | *r* | |
| /usr/local/apache2/htdocs/*.html | *r* | |
| /usr/local/apache2/logs/error_log | *rwca* | 666 |
| /usr/local/apache2/logs/access_log | *wca* | 666 |
| /usr/local/apache2/logs/referer_log | *wca* | 666 |
| /usr/local/apache2/logs/agent_log | *wca* | 666 |
| /usr/local/apache2/logs/httpd.pid | *rwc* | 644 |
| /usr/local/apache2/cgi-bin/* | *rx* | |

Fig. 4. *Partial* rules for Apache file access.

example, "$/a/b/*$" matches any file in the subtree under "$/a/b/$." Limited support for character classes (e.g., [*abc*]) is also provided.[4] A node's name can also contain environment variables and these are evaluated when the policy is being loaded into the kernel. For example, if a rule is "$/home/\${USER}/pub/*.html\ r$" and the value of *USER* is "joe," then the process will have `read` access to all HTML files under $/home/joe/pub/$. This is useful to control access of programs such as cgi-scripts which typically get most of their arguments from environment variables.

When a process makes a system call to access a file system object, the object's *fully resolved*,[5] *absolute* pathname is matched against the tree. If a path in the tree matches the object's pathname, then the access rights in the last node of the path determines if this invocation of the system call is allowed. Care has been taken to ensure that there are no security holes with the name resolution procedure. We note here that this additional resolution causes the main performance overhead in the BlueBox system. However, we note that due to this resolution the Linux operating system caches the directory entries of the all the inodes along the path. Thus the name resolution in the system call code will in most cases use these cached entries. Thus the time for resolution is not doubled. This and other issues related to performance impact of BlueBox are discussed in Section 7.

---

[4]Character ranges (e.g., [*a-h*]) and the character "]" are not allowed in a character class.
[5]A fully resolved pathname is a pathname without symbolic links, "." or "..".

The set of possible permissions on filesystem objects has been chosen to be able to easily express a wide range of policies. Besides the usual `read`, `write`, `execute`, `create`, and `append`, access rights to a file system objects also include: `delete`, `hard link to`, `soft link to`, `shared lock`, `exclusive lock`, and `truncate`. We also encode rights related to directories used as file system mount points: (a) `mount point`—a directory can be a mount point, (b) `unmount`—a file system mounted on a directory can be unmounted, and rights related to swapping devices, (c) `swapon`—a device can be a swapping device, and (d) `swapoff`—a device can be released from being a swapping device.

A node in the tree may also be associated with a list of uids and a list of gids (see Section 4.2.2). These lists are the allowed *new* user and group ownerships for file system objects matching the node.

4.2.2 *Rules for Identities.* The rules on assumable identities (uids or gids) are encoded as lists of singular integers and ranges[6] such as $[-5, -3]$, 1, $[30, 100]$, 251. The basic operation on such a list is to check if a specific integral value is in it. Each program's policy has an uid list and a gid list. These lists are the *new identities* a process running the program is allowed to assume. A process has three types of identities: *real, effective*, and *saved* [McKusick et al. 1996].[7] Since a process can freely exchange the values of different types of ids or assign one to the other, the BlueBox enforcer does not make a distinction among the three types of ids when checking the rules. In other words, when a system call requests new uids or gids, the enforcer only allows one of the following two cases:

(1) the uids/gids are in the set of uids/gids that the process already has, or
(2) the uids/gids are in the process's uid/gid list and if the following condition is met: if the process's *euid* has gone through the transition "$(euid = 0) \Rightarrow (euid = a \neq 0) \Rightarrow (euid = 0)$" and asks to change its *euid* to $v$, then $v$ equals $a$. This condition is meant to prevent an attacker from hopping over different uids.

An integer list can also represent rules on system resources with integral values such as scheduling priorities, and so on.

4.2.3 *Rules for Signals.* Rules for signals are encoded as a bit-mask,[8] which is an array of unsigned integers used as bit-vectors and represents a set of nonnegative integers whose corresponding bits are 1. Bits in a bit-mask are numbered sequentially, starting from the LSB of the first integer, numbered zero, to the MSB of the last integer. Unlike an integer list, set operations can be easily performed on bit-masks.

For rules on handling received signals, BlueBox puts signals into four subsets: (1) those *can be blocked* (CBB), (2) those *can be ignored* (CBI), (3) those *can be default* (CBD): their handlers can be the default handlers, and (4) those

[6]It may contain nonnegative and negative integers; for example, uids could be negative or nonnegative.
[7]We have currently not chosen to control change of the fsuid of a process.
[8]Bit-masks are also used to encode the *allowed system calls list*.

Table III. Socket Layers and Resources

| Layer | Resources | Access Modes/Actions |
|---|---|---|
| socket | socket file descriptors | ioctl/fcntl on aynchronous I/O, out-of-band data, ... |
| transport | <IP address, protocol, port> pairs | bind-to, connect-to, read, write, shutdown, close |
| IP | <IP address> pairs | bind-to, connect-to, read, write, close |
| routing | routing table entries | add, modify, delete/flush |
| Network Interface | Interface IP address(es) broadcast/multicast IP address(es) link addresses, ... | get/set addresses |
| | interface name | get/set Interface name |
| | mtu, ... | get/set mtu, ... |
| ARP | ARP table entries | add, modify, delete |
| Link layer | protocol specific | protocol specific |

*can be handled* (CBH): their handlers can be assigned by the process. These subsets can intersect in any possible way. Since a UNIX/LINUX system does not support other types of treatment for received signals, if a signal is in only one subset, then "can be" becomes "must be." For example, signals that are only in the CBB subset are signals that must be blocked. Besides maintaining four bit-masks for the four "can be" subsets, BlueBox also computes and maintains the *must be blocked* subset for performance reasons. An array of pointers to handlers for the CBH subset is also maintained; Section 4.3 gives more details on this array.

4.2.4 *Socket Rules.* The socket interface was originally designed for interprocess communication (IPC) across the network and within a single system [Leffler et al. 1986]. It has evolved on different flavors of UNIX since its inception and now supports functions not strictly for IPC, such as the `net_link` socket on Linux 2.2. The functionality of sockets is so rich that it is beyond the scope of this paper to discuss them and the possible rules on all of them. Our work on refining socket rules is still ongoing. We outline the general principles of our rules for sockets, using rules on sockets using protocols in the Internet Protocol Family as an example.

A socket is created by the `socket` system call which returns a file descriptor as a handle to the socket. Most of the system calls operating on file descriptors, such as `read, write, close, ioctl`, and `fcntl` can be applied to a socket file descriptor. Unlike a file, a socket is not a persistent object and vanishes when it is closed, and except for UNIX Domain sockets, a socket has no pathname. Thus we cannot directly specify file system object rules for sockets. Our principle is that a socket is just *an access point to a set of resources*. So rules on socket should be defined *in terms of access to these resources* and not on a socket that is generally transient. Table III shows some of the resources and types of accesses in the Internet Protocol Family. Resources are organized into layers, which roughly mimic the OSI 7-layer architecture. Each layer supports a specific set of functionalities and therefore specific types of access to resources in the layer.

We believe this *layering* principle applies to other types of sockets, although they may have different number of layers.

4.2.5 *Device Rules.* A device special file is the interface to a device of a specific type: such a file has a *major number* indicating the device type, and a *minor number* indicating the specific device to which the file is an interface.

Access to device special files must be carefully controlled because these files allows direct manipulation of system resources such as memory, disks, terminals, and so on. It is possible to bypass other rules by accessing devices directly: for example, if a process is allowed to read and write a disk on which a file system resides, then all rules controlling access to files on that file system can be bypassed.

The definition of the scope and semantics of BlueBox rules on device special files is still ongoing. Our main principle is that *a device special file must be controlled both as a file and as a device* for the following reasons:

—A device special file is like a regular file with a pathname, user and group ownerships, and access permissions; and it can be opened, read, written, closed, and deleted just like a regular file.
—It is necessary to prevent unintended access to a device special file, especially when its pathname does not follow the general naming tradition. Traditionally, a device special file is the offspring of the */dev* directory; and its name has a special prefix indicating the device type. For example, names of device special files for hard disks have the prefix "hd." This naming tradition is followed, but not enforced by an UNIX system. If the file */home/joe/data* is a device special file for a hard disk and a privileged process has the rule */home/joe/∗ rw* then it will be given direct read–write access to a hard disk, even if this access is not intended at all.
—Each device type also has its own unique resources and operations accessible through the `ioctl` system call, and its own unique set of ioctl commands and the corresponding parameters.

To address these concerns, we decided that the rule on a device special file will be like the rule on a regular file with the following annotations:

—A *device*: *<major number, minor number>* line to indicate this rule is for a device special file with the major and minor numbers. An access attempt to a device special file will be denied if the attempt is matched with a rule that is not annotated with the major and minor numbers of the file. So if a program has a need to access a device, the need must be *explicitly* indicated in the rule. Thus unintended access is prevented.
—An *ioctl*: *<rules on commands>* line to specify rules on the ioctl commands for this file. The syntax and semantics of "<rules on commands>" will be unique to the specific device type.

Each device type needs its own parser and kernel enforcer for its ioctl command rules. Given the fact that the number of types of devices is large and increasing but a typical computer usually has a relatively very small number

of devices installed, we decided that the kernel enforcer of a device type will be a loadable kernel module. The module is loaded when a process's rule file, which includes rules on devices of such type, is loaded into the kernel.

### 4.3 Per-Process State

Incorporating process state into rules can protect process against a much larger number of potential attacks. Several daemons, especially setuid programs, start out with real uid as root, setting only the effective uid as a user, while retaining the possibility of acquiring root state to do privileged operations. If such a daemon is subverted the attacker can then reacquire root privileges. One such example is described in the attack on the wu-ftp daemon in Section 5. Incorporation of state into the system call checks impacts performance as process state needs to be updated and checked. We have chosen to have a small amount of process state so as to minimize the performance impact. *Our guiding principle is to add state only when absolutely necessary*. Parts of the states we maintain are:

—*Identity State*: The main state component we maintain is the current process identity state. The states we note are the initial root state, user state, and reroot state when the process becomes root again. Each state has its own list of allowed system calls but other rules for the process are shared among the states. Daemons typically switch back to root state only for a short while to do a few privileged operations, and this can be effectively controlled by just changing the allowed system calls.

—*System Call Count*: Another process state component is the number of times certain system calls are made. Currently, this is enabled for only the fork and waitpid system calls. For each call we keep the current count and maximum allowed. This component is useful in two situations: first, we can use this to stop DOS attacks which repeatedly consume system resources via system calls: an attacker could repeatedly fork child processes. The second situation where this might be useful is in controlling scripts that execute arbitrary shell commands. Since the shell script forks processes to execute different commands this can control the number of commands the process can execute. While this by itself does not offer more security, it does so in combination with other rules.

—*Signal Handlers*: Another DOS attack is to have signals handled incorrectly resulting in errant process behavior. This can be done by registering a "wrong" signal handler. Since there is no way for the IDS to identify the "correct" signal handler, it assumes that the first handler registered is the right handler and does not permit any change to this.

Our philosophy to adding state to the rules is that we add state only when there is substantial benefit to be gained either in strengthening security guarantees or in making it easier to specify rules for a particular process. We note that our process state is substantially smaller than the system proposed by Sekar and Uppuluri [1999].

```
{
    ...
    /* transform http request
     * into options */
    ...
    /* Remove shell characters           * from options */
    escape_shell_command(
                    ''/sbin/ph options'');
    popen(''/sbin/ph options'','r');
    ...
}
```

Fig. 5. The PHF cgi-bin script.

### 4.4 Kernel Impact

A very important design criteria for our system was to minimize the impact on the kernel. The placement of functionality has been carefully done to reduce impact on the kernel. Our reference intrusion avoidance implementation on Linux has an intercept at the system call entry point and minor hooks in the kernel code for process creation and termination (the fork, execve, and exit system calls). The total impact on the kernel sources is limited to about 10 lines of assembly and 20 lines of C code. The rest of the enforcement process and the code to parse, allocate memory for and install rules are in a completely independent module. The patches to kernel are *very* simple and do not change the semantics of the remaining code nor do they interfere with other parts of the system. A very valid concern is the portability of BlueBox across different versions of the kernel: we believe that the points in the Linux kernel that we have intercepted are very stable and unlikely to change in revisions of the kernel. On Linux, where it is easier to allocate memory as pages, each process usually needs no more than two pages (8K) to store all IDS related structures. Of course, we use only a smaller subset of this depending on rule size and so on. We note that elementary optimization can substantially decrease this usage of kernel memory.

## 5. EXAMPLES

In this section we illustrate how our framework can be effectively used to thwart well-known attacks. They also illustrate how rules for various process can be defined.

### 5.1 Phf cgi-bin with Apache

The phf cgi-bin script was a sample script that came with the earlier distributions of Apache as an example of how cgi-bin scripts could be written. Figure 5 shows the relevant parts of the code for phf script. The script first syntactically transforms the incoming http request into a list of options for a fictional program ph and then spawns (using popen) a shell to execute ph with the created options. The `escape_shell_cmd` subroutine escapes shell characters that

may be present in the options string. The fatal bug was that it did not escape the newline (\n) character: the attack simply ensured that arbitrary command was executed by passing the command after a newline character in the options.

This is a good example of how straightforward it is to write effective rules. By design, the script invokes two commands /bin/sh (while using the popen library call) and the program /sbin/ph. Thus a very natural set of rules is to allow read and execute to these files. Besides shared libraries, the process accesses no other objects. Marking these rules as inherited ensures that the process which executes /bin/sh can only execute these two programs and the attack is thwarted. Note that the process can execute these as many times as it wants.

### 5.2 Buffer Overflow in wwwcount

The wwwcount program is a popular cgi program which maintains a count of the number of hits on a website and displays this in a graphical form. This is widely used although in nonsensitive web sites. The earlier versions of the program suffer from a well-known buffer overflow attack that can be used to execute arbitrary program on the web site. It is almost trivial to define the rules for this script. From the definition, or from an inspection of the system call audit trace for this process we can derive the proper file accesses: these are all restricted to a single directory based on the initial configuration of the program. No executable is in the rules; in fact, the execve system call is not in the allowed system call list.

### 5.3 wu-ftpd Buffer Overflow

This example illustrates how to use the state maintenance part of our system to enforce sophisticated checks. wu-ftpd is the ftp daemon developed at the Washington University at St. Louis and is one of the more popular ftp daemons in use today. There have been a number of attempts to model the behavior of the daemon to detect intrusions [Sekar and Uppuluri 1999].

At a very high level, the ftp daemon starts running as root, waits for a user to login by authentication, and sets its *effective* uid to that of the user. For the rest of this session, the daemon has as *effective* uid that of the authenticated user. It is thus in an unprivileged state, except when it needs to bind sockets to the well-known ftp data port. Since this is a privileged port, this bind operation can only be done in a privileged state so the daemon becomes root again. The only system calls made by the daemon in this state are socket, bind, and setuid to the user. Figure 6 describes this state diagram of the ftp daemon. From this functional description we can easily identify one portion of the rules for the ftp daemon. In the initial state it starts as root and is permitted to make most of the system calls; in the second state it has a nonzero uid and is permitted among other the setuid system call to become root again. In the third state the daemon is only allowed to execute the socket, bind, and setuid to user system calls. Note that this is only a subset of the entire rule set and illustrates how this thwarts a well-known attack. This subset of the rules is shown in Figure 7.

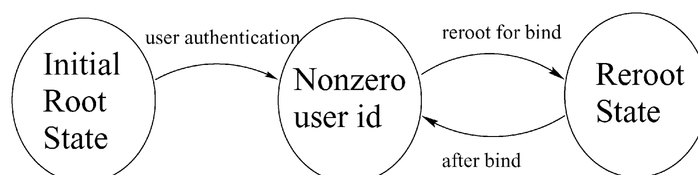


Fig. 6. State diagram of the wu-ftp daemon.

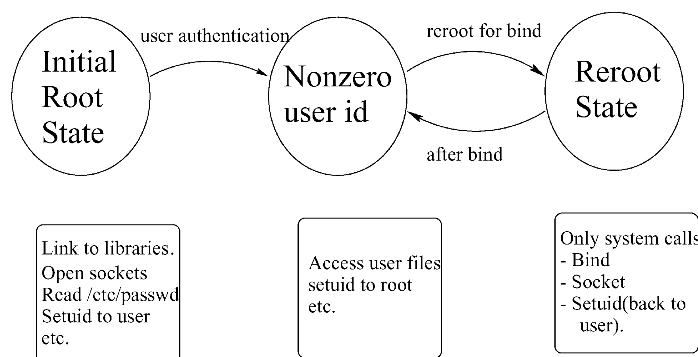


Fig. 7. Subset of the state-dependent rules for the ftp daemon.

The earlier versions of this daemon were susceptible to an attack where a regular user authenticated and overflowed the process heap [WUFTP]. Then, arbitrary code could be executed in the reroot state, for example, spawn a root shell on the server. Using the subset of the BlueBox rules described above, we can mitigate the damage due to this attack. The only system calls the attacker can execute in the reroot state are the socket, bind, and setuid to user; the attacker has no potential access to the file system objects, that is, all other sensitive system calls are disallowed. Although there is no way in the kernel, to distinguish the normal setting uid to root by the ftp daemon from the user state and the attacker setting uid to root after the buffer overflow, this is the best protection one can expect.

The examples that we have described in this section highlight several important features of the semantics of the rules in our system. They also illustrate the security guarantees the system can provide. For instance, in the case of the phf-attack, the system guarantees that the only executables are /bin/sh and /sbin/ph. However the attack can make the system endlessly execute these binaries resulting in a denial-of-service. In the ftpd example, we are unable to detect that the buffer overflowed, yet we are able to substantially mitigate the damage that the attacker can do. Another important feature is that the rules for a large number of programs are very easy to write and can potentially be done with a single examination of the audit trail. Even in the more sophisticated example of the ftp daemon, we believe our approach is substantially simpler than the state diagram-based approach advocated by Sekar and Uppuluri [1999].

## 6. EXPRESSIVE POWER OF RULES

In this section we will informally argue the soundness of BlueBox rule-checking and also describe the kinds of attacks that the rules of BlueBox will be able to protect against. We will also discuss some weaknesses of the system-call introspection approach and pieces of our extended architecture which mitigate them.

### 6.1 Rule Expressiveness

The rules of BlueBox are very comprehensive and can potentially block a large class of attacks. The richness of features in the BlueBox architecture such as state maintenance mechanisms, checks on other system calls combined with the checks on file system objects make BlueBox rules more expressive than a number of comparable systems. Some features of the rules and architecture which make the system particularly effective are as follows:

- Rules per-program. BlueBox rules are expressed on a per program basis as opposed to the per user/role approach taken by a number of extensions to operating system checks, such as role-based systems. This substantially simplifies writing rules for processes and the infrastructure in enforcing them. Substantial portions of rules are portable across systems. Making rules per program makes it substantially easier to implement very fine-grained access control. However alternatives such as role-based access control systems can yield better performance since the controls are typically well integrated into the kernel.
- State Maintenance. We have chosen to maintain a minimal amount of process state in the rules for a program. As we showed in the case of the ftp-daemon this substantially increases the number of attacks which we can protect against. Other uses for process state maintained in our system is to prevent Denial of Service type attacks since we can bound the number of times certain system calls such as fork as used.

  However, we note that maintaining state impacts performance of the system. The state we use in our system has been carefully chosen to minimize this impact.
- Parameterizable Rules. As we noted in Section 4 the rules for file system objects can be parameterized with variable names which are then instantiated with the actual values from the environment variables passed to the process upon invocation. This is particularly useful in establishing a fine-grained control of cgi-scripts which use environment variables to access file system objects.

### 6.2 TOCTTOU Attacks and Defense Against Them

A fundamental feature of the BlueBox system is that the checks on the access to resources is done at the system call entry point by introspection of the system call parameters. This is to be contrasted with the approach of the Linux Security Module [LSM] where checks on the access to resources is done at various points

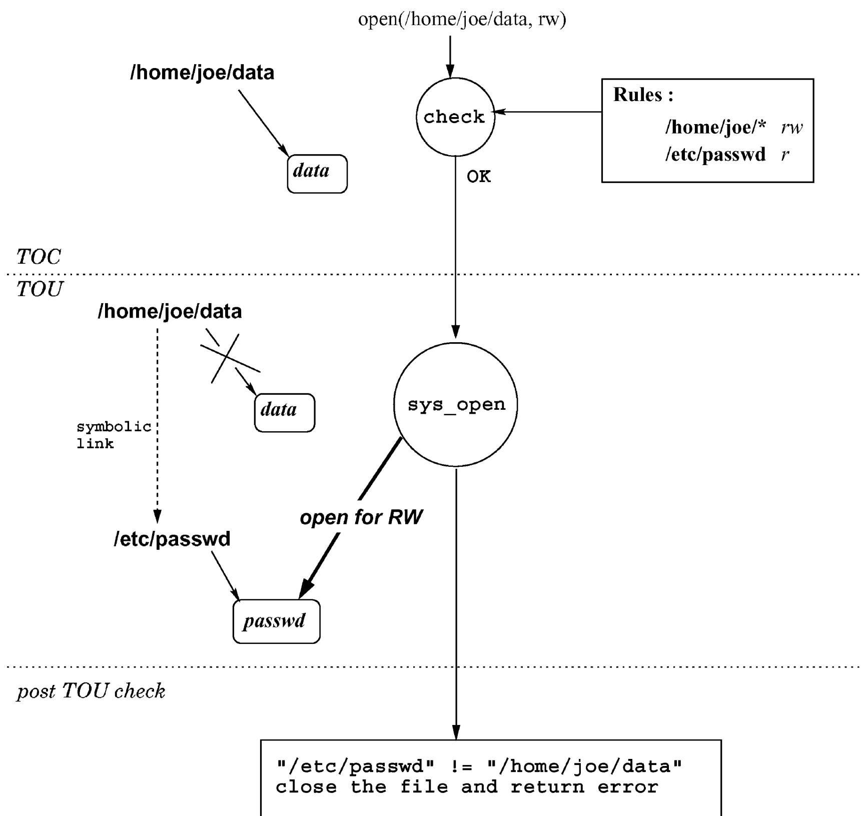


Fig. 8. TOCTTOU attack on file system objects and defense against it.

in the kernel, close to where the resources are actually accessed. While the BlueBox approach is less intrusive on the kernel code, by design the checks are made much before the point where the resources are accessed. This time interval between check and actual access can be exploited maliciously to mount what are called *time-of-check-to-time-of-use* ( TOCTTOU ) attacks [Bishop and Dilger 1996]. A TOCTTOU attack can happen if a parameter of a system call invocation can be made to refer to one system object at the time of check but to another system object at the time of use.

Figure 8 shows a TOCTTOU attack on file system objects and the possibilities to defend against these attacks. Assuming the rules for a process running as root allow read—write access to all files under the directory */home/joe* but only read access to */etc/passwd*. The process makes an open system call to open a file with the pathname */home/joe/data* for reading and writing. This call passes the BlueBox check. Before the file is actually opened, the link between the pathname and the file is severed and the pathname is made to refer to a symbolic link pointing to */etc/passwd*. Therefore */etc/passwd* will be opened for read—write; this is a clear violation of the rules.

There are three options to mitigate TOCTTOU attacks on file system objects:

(1) Conduct the checks at time of use. This option is not directly applicable and was rejected because it violates the "wrapper" design principle and it would lead to a big impact on the kernel.

(2) Change the syntax and semantics of the rules on file system objects so such a rule will refer to the object directly, not to the pathname pointing to the object. This would mean referring to an file system object by its device number and inode number [McKusick et al. 1996]. This option has substantial limitations including those listed below and was hence not adopted.

  — It is counterintuitive. Most users and system administrators do not access files through their device numbers and inode numbers.
  — It could not support wildcard in rules.
  — Different versions of a file may have different inode numbers. For example, if the popular text editor `emacs` is used to modify the file */etc/passwd*, then the original file will be renamed to */etc/passwd~*, and a new version of the file will be created with the same pathname */etc/passwd* but a different inode number than that of the original file.

(3) The third option we considered was to do another check after the time of use, at the end of the system call, to see if *the same* file was checked upon and accessed. We have decided to use this option since it keeps the impact on the kernel minimal and yet retains all the flexibility of our approach. However, we note that in some cases the effect of a system call may not be easy to reverse.

To determine if the same file is checked upon and accessed, the fully resolved pathname that is used to check against the rules, and the type of the file (regular, directory, device special, and so on) are recorded at the time of check. After the object is accessed, the object's fully resolved pathname and type is compared with the records and a mismatch indicates a TOCTTOU attack. In the case of the `open` system call, the opened file is closed and an error is returned. To improve performance, we implement a shortcut for detecting TOCTTOU attacks on file system objects. At the time of check, a reference is placed on the inode of the file being checked; after the time of use, we compare the inode of the accessed object to the inode referenced at the time of check. If they are not the same, the file name and type matching is performed. The reference holds the inode in system memory and is released after the inode comparison.

## 7. PERFORMANCE

One of the main design guidelines for BlueBox is to minimize the performance impact. Crucial design decisions about how much state to incorporate into the rules were driven primarily by how much it impacts the performance of the process being monitored. The prototypical application we use to measure the performance is the Apache 2.0 web server daemon. The results for this daemon are representative as it exercises most of the checks implemented for the various

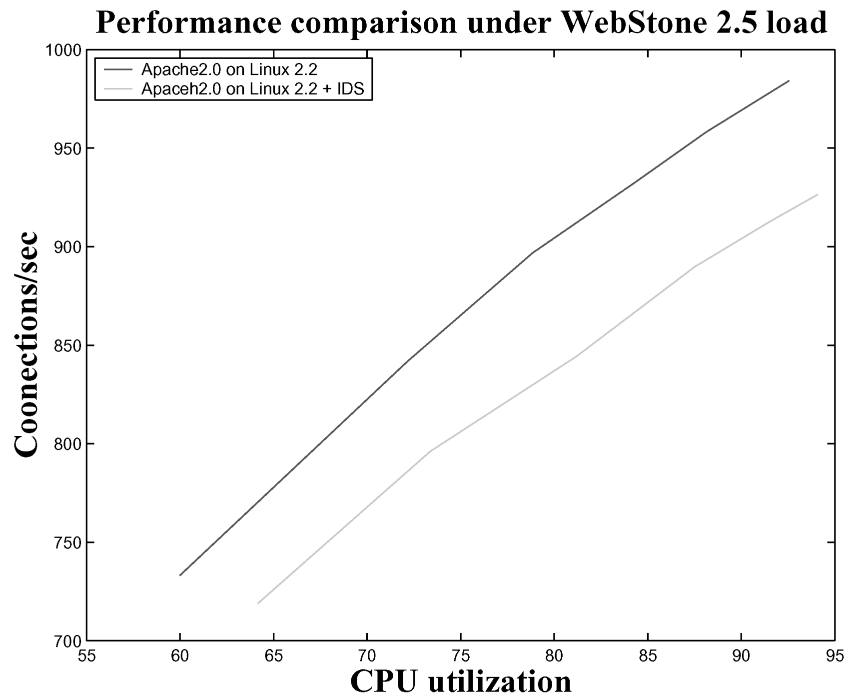


Fig. 9. Performance of the Apache 2.0 with and without system call checks.

system calls. In fact, many of the compute intensive system call checks, such as open, read, and fcntl, are used substantially. Other processes will typically use fewer such calls, and hence the performance impact on the Apache httpd daemon will be an upper bound.

## 7.1 Testbed

Our tests ran the WebStone benchmark of server performance with the following parameters: there is a single client machine generating load and it has between three and eight threads generating requests for the server. These were so chosen such that the resulting load does not saturate the server with or without BlueBox. The load generated by the clients is entirely static content. Testing under dynamic content would result in a larger penalty due to the overhead of loading rules for each script that is invoked. Both the webstone client and the Apache server were put on a gigabit ethernet to ensure that no effect due to large network latencies were observed in the results.

## 7.2 Test Results

Figure 9 shows the performance of the Apache 2.0 webserver performing with and without BlueBox under various server load factors. We anticipate a 8–10% performance penalty for the Apache 2.0 server running on the Linux 2.2.14 kernel.

### 7.3 Bottlenecks

The main performance bottlenecks in enforcing the system call checks for the Apache server is pathname resolution. For each request, the Apache server opens a file and then uses sendfile to send it over the socket. For each request, we perform a full name resolution operation to match the right file name with a node on the tree of file system object rules to eliminate security holes. This can be additionally optimized by caching and marking certain names as fully resolved. Systems such as LSM/SeLinux, have *mandatory access control* type labels [DoD 1985] on file system objects and move the check entirely to the file system, that is, the file system will check the labels for permission before it opens the file. Although this may yield better performance, we have not adopted this in BlueBox since this would result in a very large impact on kernel code.

The results shown in Figure 9 were generated entirely using static content. Dynamic content requires the server to load another process and thus load the rules for this new process that adds to the performance penalty. This can be somewhat mitigated by caching the data structures representing rules for frequently used cgi-bin scripts. We are in the process of implementing this in the BlueBox implementation on Linux.

Using these optimizations, we expect that the performance penalty for the Apache daemon will be close to 5%. We believe that this penalty is not excessive given the security guarantees one can obtain using this system.

## 8. FUTURE RESEARCH DIRECTIONS

Besides the ongoing work of improving performance and refining the syntax and semantics of rules, research effort in our system will focus on addressing the following issues:

—*Rule generation aids* to make the generation of program rules more easier. We think that work on static analysis of programs [Wagner and Dean 2001; Ashcraft and Engler 2002; Zhang et al. 2002] is a promising approach.

—A *flexible response* mechanism to allow different levels of responses to an attack based on the program and the resources being attacked, policy configuration, and other inputs from the surrounding environment.

—A *report* mechanism to report detected attacks and responses taken.

—Using the report and response mechanisms as links to integrate the three different types of IDS, namely, misuse detection, anomaly detection, and the policy-based approach into an *integrated defense system*.

### 8.1 Integrated Defense System

We believe BlueBox is very good at detecting attacks. However, BlueBox *cannot* do the following by design:

—*Identifying the attack*: BlueBox can detect a rule is violated, but it cannot tell which attack is being carried out.

—*Early detection*: BlueBox can only detect an attack after the start of the execution of the attack.

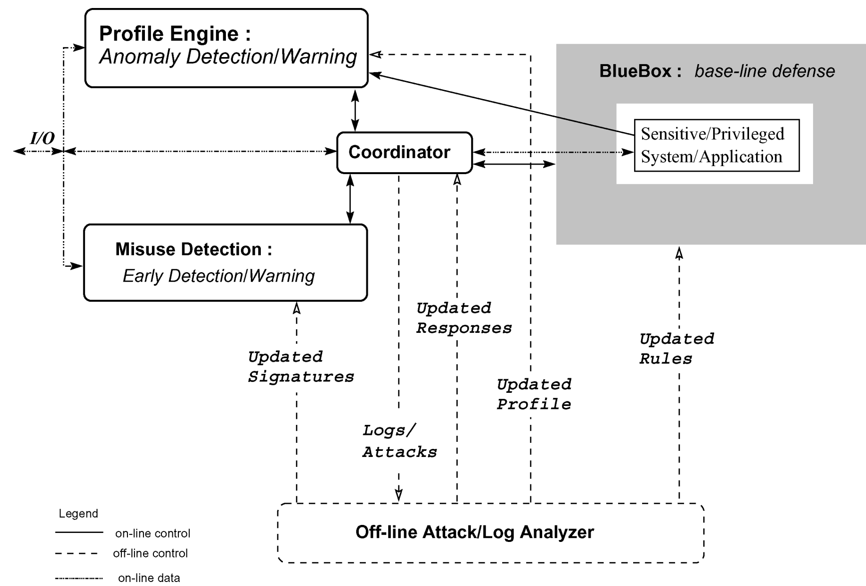


Fig. 10. Integrated defense system.

Misuse detection can identify attacks and can detect attacks before the start of their executions on a system but its false positive and negative rates are usually high. Anomaly detection can detect attacks disguised as legitimate activities, such as access to a sensitive file at an usual time or an unusually high level of a legitimate activity, but the detection may have to be done off-line due to the need to collect statistics and incorrect detection rates could be high. Each of the three approaches has unique abilities that can compliment the other two. So we submit that the best defense is to integrate the three approaches together.

Figure 10 depicts our current thinking of an integrated defense system. Like a nation protects itself, the misuse detection box serves as an *intelligence agency* watching out for attacks from the outside, anything it misses is subject to BlueBox rule checking which servers as the *baseline defense* at the boundary of a system or an application. Any attacks disguised as legitimate activities and which pass BlueBox is subject to the watchful eyes of the profile engine which is like a *law-enforcement agency* watching for unusual, suspicious activities. The *coordinator* uses information provided by the three approaches to make *on-line* decisions regarding responses to attacks; it may also change BlueBox's responses to attacks.

The *attack/log* analyzer correlates and analyzes the system/network logs and detected attacks; its goal is to combine information collected by the three approaches to enhance them. The analyzer can identify signatures of previously unknown attacks, refines signatures of known attacks, adjusts the behavior of the profile engine, refines the rules of BlueBox, and so on. Due to the difficulty of analyzing large amount of logs and attacks, we expect the analyzer to work *off-line* and it may very well need human assistance.

## 9. CONCLUSION

We have presented BlueBox, a simple system for sandboxing applications that can substantially mitigate security exposures of processes. We believe that our system is a simple and comprehensive way to incorporate checks on the execution of programs at the time of invocation of system calls. We have described rules for important servers such as the Apache daemon and a number of popular cgi-bin scripts; these rules can be used as templates across installations with new rules written for the individual scripts. Our rule syntax and semantics are simple and yet quite effective in catching a large number of known attacks. Since performance has been a motivating factor in our design, we have achieved our security guarantees with minimal impact on the performance.

On a much larger scale, we believe that much more effective security can be achieved by combining the attack signature-based systems, statistical profile-based systems, and the sandboxing systems such as the one described in this paper into a integrated defense system.

## APPENDIX: BLUEBOX KERNEL MODULES ON LINUX

To avoid any changes to the existing Linux 2.4.18 kernel code, we implemented BlueBox using two loadable kernel modules:

*System Call Interception Module*. This module intercepts every system call made by a process. If the process is being monitored, the control is then transferred to the *Rule Enforcement Module* which is *registered* with the *System Call Interception Module*, otherwise the control is transferred back to the kernel's handler routine of the particular system call.

*Rule Enforcement Module*. This module *registers* itself with the *System Call Interception Module* and checks a process's invocation of a system call and its parameters against the BlueBox rules associated with a process. An error is returned if the rules are violated, otherwise the control is transferred back to the kernel's handler routine of the particular system call.

We believe the important concept here is the separation of the system call interception mechanism from the operation performed on an intercepted system call. BlueBox rule enforcement is just one of many possible operations to perform on intercepted system calls; other possibilities include detailed auditing, statistical profiling, etc. We actually implemented a kernel version of the *strace* utility to audit system calls. This auditing mechanism can monitor multiple processes simultaneously and provided us great help in understanding a process's behavior. In particular, this auditing mechanism does not interfere with delivery of signals as *strace* does so it will not block the operation of a group of processes communicating through signals.

### ACKNOWLEDGMENTS

This work has benefited substantially from discussions with a large number of people. In particular, we would like to acknowledge the contributions of Pankaj Rohatgi, Josyula R. Rao, David Safford, and Douglas Schales of IBM Research, and Hervé Debar who was at IBM Zurich Research Lab. Our summer intern

Oleg Kolesnikov implemented a primitive report-and-response subsystem. We would like to give special thanks and praises to our summer intern Ramkumar ChinChani who did a very good job of porting BlueBox to Linux 2.4.18 and moving the entire rule enforcer into a loadable kernel module.

## REFERENCES

ANDERSON, D., LUNT, T. F., JAVITZ, H., TAMARU, A., AND VALDES, A. 1993. SAFEGUARD FINAL REPORT: Detecting unusual program behavior using the NIDES statistical component, Tech. Rep., Computer Science Laboratory, SRI International, Menlo Park, CA, USA.

ASHCRAFT, K. AND ENGLER, D. 2002. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*.

ATKINSON, R. 1995. Security architecture for the Internet protocol. Internet RFC 1825.

BERNASCHI, M., GABRIELLI, E., AND MANCINI, L. 2000. Enhancements to the Linux kenel for blocking buffer overflow based attacks. Available at `http://www.iac.rm.cnr.it/newweb/tecno/papers/bufoverp`.

BERNASCHI, M., GABRIELLI, E., AND MANCINI, L. V. 2002. REMUS: A security-enhanced operating system. *ACM Trans. Inf. Syst. Sec. 5*, 1 (February).

BISHOP, M. AND DILGER, M. 1996. Checking for race conditions in file accesses. *Computing Syst. 9*, 2, 131–152.

CHARI, S. N. AND CHENG, P. C. 2002. BlueBox: A policy-driven, host–based intrusion detection system. In *Network and Distributed System Security Symposium*.

COWAN, C., BEATTIE, S., KROAH-HARTMAN, G., PU, C., WAGGLE, P., AND GLIGOR, V. 2000. SubDomain: Parsimonious server security. In *Proceedings of the 14*th *Systems Administration Conference (LISA 2000 )*.

CROSBIE, M., DOLE, B., ELLIS, T., KRSUL, I., AND SPAFFORD, E. 1996. IDIOT users guide. Tech. Report CSD-TR-96-050, COAST Laboratory, Dept. of Computer Sciences, Purdue University.

DEBAR, H., DACIER, M., NASSEHI, M., AND WESPI, A. 1998. Fixed vs. variable-length patterns for detecting suspicious process behavior. Research Report, No. RZ3012, IBM Research Division, Zurich Research Lab.

DEBAR, H., DACIER, M., AND WESPI, A. 1999. Towars a taxonomy of intrusion detection systems. *Computer Networks 31*.

DIERKS, T. AND ALLEN, C. 1997. The TLS protocol version 1.0. IETF <draft-ietf-tls-protocol-02.txt>.

DoD 1985. US Department of Defense trusted computer system evaluation criteria. DOD 5200.28-STD. Available at `http://www.radium.ncsc.mil/tpep/library/rainbow/index.html`.

ERLINGSSON, Ú. AND SCHNEIDER, F. B. 2000. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*.

FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. 1996. A Sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*.

FRASER, T., BADGER, L., AND FELDMAN, M. 1999. Hardening COTS software with generic software wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*.

FREIER, A. O., KARLTON, P., AND KOCHER, P. C. 1996. The SSL protocol version 3.0. IETF <draft-ietf-tls-ssl-version3-00.txt>.

JACKSON, K. A. 1999. Intrusion Detection System (IDS) product review. IBM internal confidential document, IBM Research Division, Zurich Research Lab.

JAIN, K. AND SEKAR, R. 2000. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed Systems Security Symposium*.

JAVITZ, H. AND VALDES, A. 1994. The NIDES statistical component description and justification. Tech. Rep., Computer Science Laboratory, SRI International, Menlo Park, Cal., USA.

JVM 2001. The Java virtual machine. Available at `http://www.javasoft.com`.

KO, C., FRASER, T., BADGER, L., AND KILPATRICK, D. 2000. Detecting and countering system intrusions using software wrappers. In *Proceedings of the 9th USENIX Security Symposium*.

LEFFLER, S. J., JOY, W. N., FARBY, R. S., AND KAREL, M. J. 1986. Networking implementation notes, 4.3BSD edition. In *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX–11 Edition*. USENIX Association.

LSM. The Linux Security Module Project. Code available at http://lsm.immunix.org.

MCKUSICK, M. K., BOSTIC, K., KARLES, M. J., AND QUARTERMAN, J. S. 1996. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, New York City, USA, 67, 540.

PAXSON, V. 1998. Bro: A system for detecting network intruders in real-time. In *the 7th USENIX Security Symposium*.

PTACEK, T. H. AND NEWSHAM, T. N. 1998. Insertion, evasion, and denial of services: Eluding network intrusion detection. Available at http://www.nai.com.

RANUM, M. J., LANDFIELD, K., STOLARCHUK, M., SIENKIEWICZ, M., LAMETH, A., AND WALL, E. 1997. Implementing a generalized tool for network monitoring. In *the 11th USENIX Systems Administrator Conference*.

SEKAR, R. AND UPPULURI, P. 1999. Synthesizing fast intrusion detection systems from high-level specifications. In *the 8th USENIX Security Symposium*, 63–78.

SELINUX. Security-enhanced Linux. Available at http://www.nsa.gov/selinux/.

WAGNER, D. A. 1999. Janus: An approach for confinement of untrusted applications. Tech. Rep. CSD–99–1056, University of California at Berkeley.

WAGNER, D. A. AND DEAN, D. 2001. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*.

WALKER, K. M., STERNE, D. F., BADGER, M. L., PETKAC, M. J., SHERMANN, D. L., AND OOSTENDROP, K. A. 1996. Confining root programs with domain and type enforcement. In *the 6th USENIX Security Symposium*.

WEBSPHERE 2001. WebSphere V4.0 Advanced Edition Handbook. Available at http://www.redbooks.ibm.com/redpieces/pdfs/sg246176.pdf.

WUFTP. Source code to exploit the heap overflow in wu–ftpd. Available at http://oliver.efri.hr/~crv/ security/bugs/mUNIXes/wuftpd15.html.

XIE, H. AND BIONDI, P. 2001. The Linux Intrusion Detection Project. Available at http://www.lids.org.

ZHANG, X., EDWARDS, A., AND JAEGER, T. 2002. Using CQUAL for static analysis of authorization hook placement. In *the 11th Usenix Security Symposium*.

Received May 2002; revised December 2002; accepted December 2002