BlueBoX: A Policy–driven, Host–Based Intrusion Detection system

Suresh N. Chari Pau–Chen Cheng

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

{schari,pau}@watson.ibm.com

Abstract

In this paper we describe our experiences with building BlueBox, a host based intrusion detection system. Our approach 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 2.2.14 kernel, and have built rule templates for popular daemons such as Apache 2.0 and wu-ftpd. We are validating our design 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 will discuss the motivation and rationale behind BlueBox, its design, implementation on Linux, and related work.

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 controls the flow of through communication and network IDSs detect possible attacks by monitoring the communication. While firewalls, when properly configured, serve their intended purpose, current network IDSs suffer from a number of limitations. Network IDSs typically analyze traffic on the network and either scan for patterns containing known attacks or detect statistically abnormal patterns. With the advent of traffic encryption protocols such as SSL [FKK96, DA97] and IPSEC [Atk95], a significant portion of traffic on the Internet is encrypted and therefore is unavailable for examination. Also, there are well–known ways to evade network IDSs [PN98]. 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.

To contrast our approach we first look at mechanisms used in currently deployed host based IDSs. They are primarily based on one of the following [DDW99, Jac99]:

- Anomaly detection : Defined by a statistical profile of "normal" behavior [JV94, ALJ$^+$93, FHSL96, DDNW98]. A pattern that deviates significantly from the normal profile is considered an attack.
- Misuse detection: Defined by collections of signatures of known attacks [Jac99, Pax98, RLS$^+$97, CDE$^+$96]. Activities matching such patterns are considered attacks.

Conceptually, misuse detection is based on knowledge of bad behaviors (attacks) and anomaly detection is based on knowledge of good (normal) behaviors. If both techniques were perfect, then each would exactly complement the other: *i.e.* 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 behaviors as attacks. Likewise, anomaly detection can not cover all good behaviors and will mistake some attacks for good behaviors. 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 for an attacker to gradually increase its level of malicious activities until these activities are considered normal.

Our policy–driven technique, like the concept of *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 3 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 which 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 2.

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 has two advantages: (1) *unknown attacks* can be detected, (2) previously unseen but legitimate behaviors would not be mistaken for attacks. Therefore the false positive and (hopefully) the 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 the rules are precisely defined in terms of system resources and not by attacks, 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 there is the option to block an unauthorized access or act.

On the other hand, since the rules are defined on access to system resources there are disadvantages as compared to other IDSs. Some of them are:

- 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' can not be detected.

The rest of the paper is organized as follows: Section 2 surveys related work and compares them with our system. Section 3 gives an overview of the specification and generation of rules. 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 details experiences on specifying rules. Section 6 discusses the performance impact of the IDS and we conclude in Section 7.

2 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.

2.1 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 [JVM01]. 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:

2.1.1 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 [UES00]. 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 in to 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.

2.1.2 Type based mechanisms

The celebrated Java Virtual Machine is a classic example of a system which enforces strong checks on 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.

2.2 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 [FHSL96, DDNW98]. The main advantage of these systems is the minimized impact on the kernel *i.e.* one needs to make few changes to the kernel to implement them. However, they can not 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.

2.3 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.

2.3.1 LIDS

The Linux Intrusion Detection system (LIDS) [XB01] 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 this. 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.

2.3.2 A Program as a Finite State Machine

Sekar *et al* [SU99] 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.

2.3.3 Generic software wrappers

Generic software wrappers[KFBK00, FBF99] 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 which 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.

2.3.4 Other Sandboxing Systems

The system that comes closest to our system is the work of Bernaschi *et al* [BGM00]. Their 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 by Walker *et al* [WSB+96] 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 non–file–system–object resources and seems to incur more complexity when providing fine–granularity control than BlueBox.

2.4 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 [Wag99, JS00]. 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[JS00], 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 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.

3 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 1.

Features of our current rule specification includes:

- Access permissions to file system objects.
- Access to the file system, e.g., mount, unmount.
- Permitted *uid* and *gid* transitions.
- signals which can be sent, received, blocked, ignored & handled.
- Process characteristics such as scheduling priorities which can be modified.
- Elementary controls for other system resources such as IPC objects, sockets and ioctl calls. This is an area requires more study for more comprehensive rules.

To make the policy specification 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 don't 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. 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 Appendix A.

The policy for a program can also be marked *inheritable*: this is useful for a script where each program executed by the script can share the script's policy.

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 through 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 which 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 which 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[WEB01].

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.

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 1. 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.

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, $\cdots$)	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

Table 1. types of resources and access

Policy Loading and Enforcement Since BlueBox policies are meant to control access to system resources which can only be accessed through system calls, the natural place for rule enforcement is at the kernel system call entry point. Our prototype on Linux 2.2.14 places an *enforcer* module at the kernel system call entry point to enforce rules. 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.

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 [1] 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*; i.e., 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[2] to hold its private BlueBox state which 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.

[1]The API for execve is not changed.
[2]At present, the size of the buffer is one page or $4K$ bytes.

4.2 Rules for Different Types of Resources

In this section we will discuss rules for three types of resources, namely file system objects, uid/gid lists and signals; each has particular syntax and semantics. We believe the syntax and semantics discussed here can represent most, if not all, of BlueBox rules.

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 2 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 "$*$" repre-

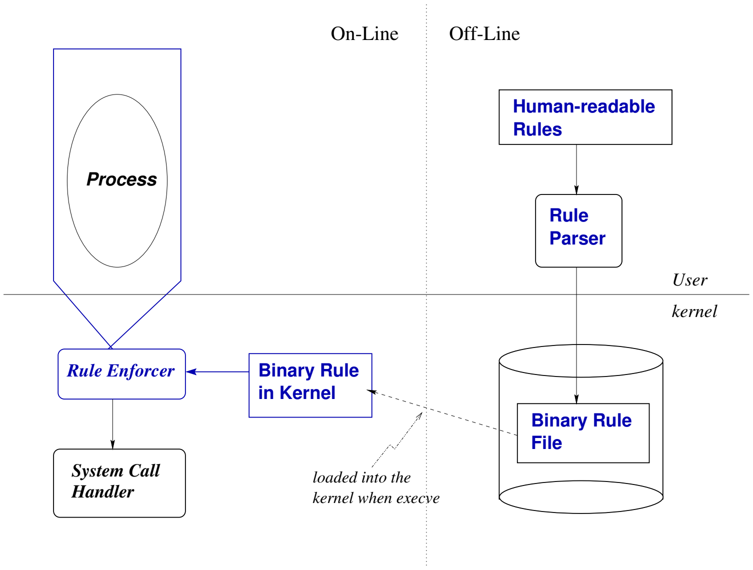

Figure 1. BlueBox Architecture

sents an entire subtree; for example, "$/a/b/*$" matches any file in the subtree under "$/a/b/$". Limited support for character classes (e.g., $[abc]$) is also provided[3]. 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 $/home/joe/pub/$. When a process makes a system call to access a file system object, the object's *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.

Besides the usual `read, write, execute, create, append`, access rights to a file system objects also include : `delete, hard link to, soft link to, shared lock, exclusive lock, truncate`. There are also 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, (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 uid's and a list of gid's (see Section 4.2.2). These lists are the allowed *new* user and group ownerships for file system objects matching the node.

[3]Character ranges (e.g., $[a\text{-}h]$) and the character '] ' are not allowed in a character class.

4.2.2 Rules for Identities

Rules on identities (uids or gids) are encoded as lists of singular integers and ranges[4] such as $[-5,-3]$, 1, $[30,100]$, 251, 300. 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* [MBKQ96]. 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 id's when checking the rules. In other words, when a system call requests new uid's or gid's, the enforcer only allows one of the following two cases :

1. the uid's/gid's are in the set of uid's/gid's which the process already has, or

2. the uid's/gid's 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 uid's.

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

[4]It may contain non–negative and negative integers; e.g., uid's could be negative or non–negative.

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*	

Figure 2. *Partial* rules for Apache file access

4.2.3 Rules for Signals

Rules for signals are encoded as a bit–mask[5], which is an array of unsigned integers used as bit–vectors and represents a set of non–negative 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, (4) those *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.

[5]Bit–masks are also used to encode the *allowed system calls list*.

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 re–acquire 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. For each state, there is a separate edition of the rules dictating which system calls are allowed but all states share the same set of file system access permissions. 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: *e.g.* an attacker could repeatedly fork child processes. The second situation where this might be useful is in controlling scripts which 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 if 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 *et. al* [SU99].

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 which 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 2 pages (8K) to store all IDS related structures. Of course, we use only a smaller subset of this depending on rule size etc. Substantial portions of the rules are shared by processes and any child thread/process that they spawn. This can be reduced with elementary optimization.

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 which came with the earlier distributions of Apache as an example of how cgi–bin scripts could be written. Figure 5.1 shows the relevant parts of the code for phf script. The script first

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

Figure 3. The PHF cgi–bin script

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 which 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 new 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 non sensitive web sites. The earlier versions of the program suffer from a well known buffer overflow attack which 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 [SU99].

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 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 user. Figure 4 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 5.

The earlier versions of this daemon were susceptible to an attack where a regular user authenticated and overflew the process heap[WUF]. Then, arbitrary code could be executed in the reroot state *e.g.* 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 *i.e.* 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 [SU99].

6 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 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.

6.1 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

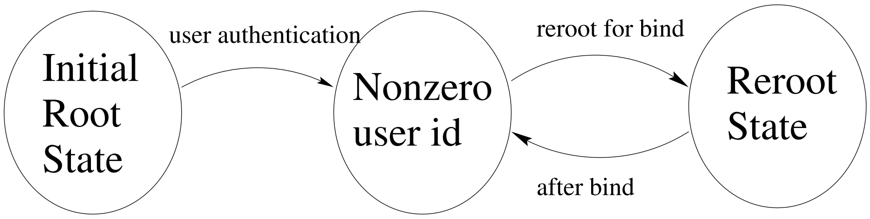

Figure 4. State diagram of the wu–ftp daemon

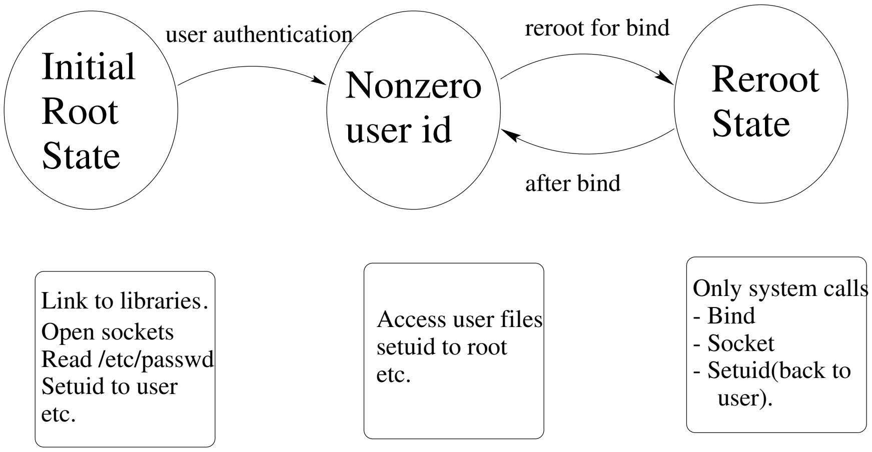

Figure 5. Subset of the state–dependent rules for the ftp daemon.

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.

6.2 Test Results

Figure 6 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.

6.3 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. Another way to reduce this overhead of name resolution is to have *mandatory access control* type labels [DoD85, SEL] on file system objects and move the check entirely to the file system i.e. the file system will check the labels for permission before it opens the file.

The results shown in Figure 6 were generated entirely using static content. Dynamic content requires the server to load another process and thus load the rules for this new process which 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.

7 Conclusion

We have presented Blue Box, a simple system for sandboxing applications which 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

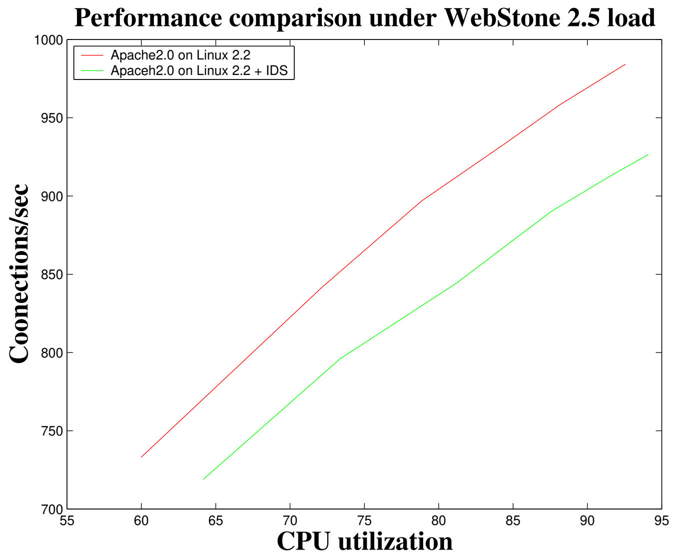

Figure 6. Performance of the Apache 2.0 with and without system call checks

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 integrating the attack signature based systems, statistical profile based systems and the sandboxing systems such as the one described in this paper. Depicted in Figure 7, the signature based approach detects attacks from the outside, the statistical profile approach detects anomaly inside, and the sandboxing approach stops attacks on the boundary.

8 Acknowledgements

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.

References

[ALJ+93] Debra Anderson, Teresa F. Lunt, Harold Javitz, Ann Tamaru, and Alfonso Valdes. SAFEGUARD FINAL REPORT: Detecting Unusual Program Behavior Using the NIDES Statistical Component. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA, December 1993.

[Atk95] Randall Atkinson. Security Architecture for the Internet Protocol. Internet RFC 1825, August 1995.

[BGM00] M. Bernaschi, E. Gabrielli, and L. Mancini. Enhancements to the Linux Kenel for Blocking Buffer Overflow Based Attacks, `http://www.iac.rm.cnr.it/newweb/tecno/papers/bufoverp`, August 2000.

[CDE+96] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT Users Guide. Technical Report CSD-TR-96-050, COAST Laboratory, Dept. of Computer Sciences, Purdue University, September 1996.

[DA97] Tim Dierks and Christopher Allen. The TLS Protocol Version 1.0. IETF <draft-ietf-tls-protocol-02.txt>, March 1997.

[DDNW98] Hervé Debar, Marc Dacier, Mehdi Nassehi, and Andreas Wespi. Fixed vs. Variable–Length Patterns for Detecting Suspicious Process Behavior, Research Report, No. RZ3012, IBM Research Division, Zurich Research Lab, April 1998.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towars a taxonomy of intrusion detection systems. *Computer Networks*, 31, 1999.

[DoD85] US Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD,

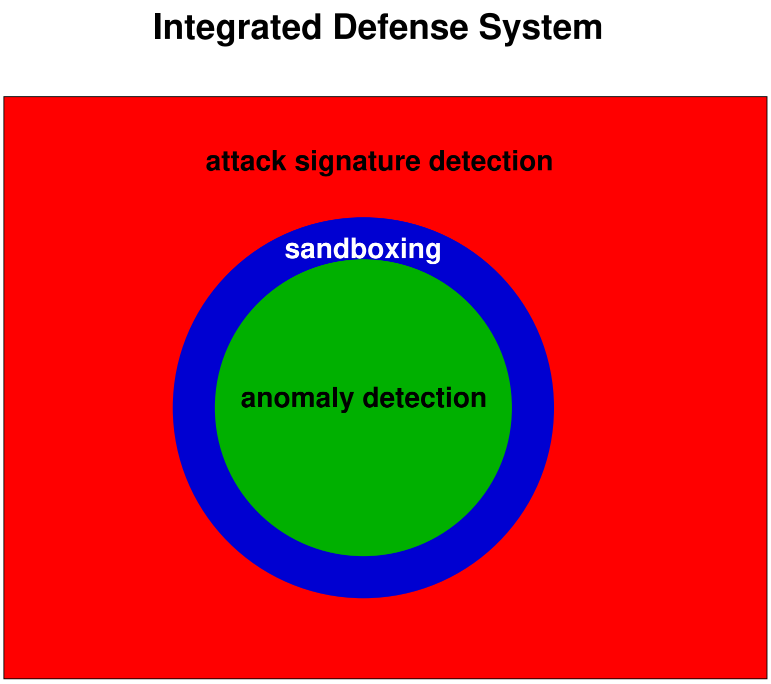

Figure 7. Integrated Defense

http://www.radium.ncsc.mil/tpep/library/rainbow/index.html, December 1985.

[FBF99] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for UNIX Processes. In *IEEE Symposium on Security and Privacy*, 1996.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol Version 3.0. IETF <draft-ietf-tls-ssl-version3-00.txt>, November 1996.

[Jac99] Kathleen A. Jackson. INTRUSION DETECTION SYSTEM (IDS) Product Review, IBM internal confidential document, IBM Research Division, Zurich Research Lab, April 1999.

[JS00] K. Jain and R. Sekar. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2000.

[JV94] Hal Javitz and Alfonso Valdes. The NIDES statistical component description and justification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, USA, March 1994.

[JVM01] The Java Virtual Machine, http://www.javasoft.com, 2001.

[KFBK00] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and Countering System Intrusions Using Software Wrappers. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karles, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*, pages 67, 540. Addison Wesley, New York City, New York, USA, 1996.

[Pax98] Van Paxson. Bro: A System for Detecting Network Intruders in Real–Time. In *the 7th USENIX Security Symposium*, 1998.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Services: Eluding Network Intrusion Detection, http://www.nai.com, January 1998.

[RLS+97] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lameth, and Eric Wall. Implementing a Generalized Tool for Network Monitoring. In *the 11th USENIX Systems Administrator Conference*, 1997.

[SEL] Security–Enhanced Linux. Available online at `http://www.nsa.gov/selinux/`.

[SU99] R. Sekar and P. Uppuluri. Synthesizing Fast Intrusion Detection Systems from High-Level Specifications. In *the 8th USENIX Security Symposium*, pages 63–78, August 1999.

[UES00] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[Wag99] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD–99–1056, University of California at Berkeley, August 1999.

[WEB01] WebSphere V4.0 Advanced Edition Handbook. Online at `http://www.redbooks.ibm.com/redpieces/pdfs/sg246176.pdf`, November 2001.

[WSB+96] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendrop. Confining Root Programs with Domain and Type Enforcement. In *the 6th USENIX Security Symposium*, July 1996.

[WUF] Source code to exploit the heap overflow in wu–ftpd. Online at `http://oliver.efri.hr/~crv/security/bugs/mUNIXes/wuftpd15.html`.

[XB01] Huagang Xie and Philippe Biondi. The Linux Intrusion Detection Project, `http://www.lids.org`, 2001.

Appendix

A Harmless System Calls

Each of these system calls either has no security implications or is not supported by the Linux 2.2.14 kernel.

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

Table 2. *Harmless* System Calls