Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies

Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman and Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Laboratory
Department of Computer Science and Engineering,
The Pennsylvania State University
{hvijay,ggj102,ruedarod,jschiffm,tjaeger}@cse.psu.edu

ABSTRACT

Adding new programs or configuration options to a system often leads to new exploits because it provides adversaries with new ways to access possible vulnerabilities. As a result, application developers often must react to exploits as they are found. One proactive defense is to protect programs at their *attack surfaces*, the program entry points (e.g., system calls) accessible to adversaries. However, experience has shown that developers often fail to defend these entry points because they do not locate all such system calls where programs access system resources controlled by attackers. In this paper, we develop a runtime analysis method to compute program attack surfaces in system deployments, which uses a novel approach to computing program adversaries to determine which program entry points access adversary-controlled objects. We implemented our design as a Linux kernel mechanism capable of identifying entry points for both binary and interpreted programs. Using this mechanism, we computed the attack surfaces for all the programs in the Ubuntu Linux 10.04 Desktop distribution automatically. On examining located attack surfaces, we discovered previously unknown vulnerabilities in an X Windows startup script available since 2006 and the GNU Icecat web browser. Our tools enable developers to find attack surfaces for their programs quickly and to produce defenses prior to the emergence of attacks, potentially moving us away from the penetrate-and-patch rut.

1. INTRODUCTION

Protecting host system integrity in the face of determined adversaries remains a major problem. Despite advances in program development and access control, attackers continue to compromise systems forcing security practitioners to regularly react to such breaches. With the emergence of more sophisticated malware, such as Stuxnet, malware has begun to target program entry points that are left undefended, thus exacerbating the problem.

While security practitioners may eventually learn which entry points must be defended over a software's lifetime, new software and configuration options are frequently introduced, opening additional vulnerabilities to adversaries. The application developers' problem is to identify the program entry points accessible to adversaries and provide necessary defenses at these entry points before the adversaries use these to compromise the program. Unfortunately, this is a race that developers often lose. While some program vulnerable entry points are well-known (mostly network), the complexity of host systems makes it difficult to prevent local exploits should attackers gain control of any unprivileged processing. For example, the OpenSSH daemon was reengineered to defend two entry points in the privileged part through which several vulnerabilities were exploited [25], but a third entry point also existed that was vulnerable to any user processes [28]. The question we explore in this paper is whether the program entry points accessible to adversaries can be found proactively, so defenses at these entry points can also be developed proactively.

Prior efforts to better understand how adversaries can access programs focus either on system security policies or program entry points, but each provide a limited view. With the widespread introduction of mandatory access control (MAC) enforcement in commercial operating systems (OSes) [36, 32, 35], it is possible to determine the subjects in the MAC policy that may be influenced by adversary-controlled data [34, 7, 14, 31]. Also, methods have been developed to compute *attack graphs* [30, 23, 20], which generate a sequence of adversary actions that may result in host compromise. However, these methods treat programs as black boxes, where any program entry point may be able to access either adversary-controlled data or benign data. As these accesses are not connected to the program entry points that use them, it is difficult to know where exactly in the program or even the number of points in the program that access adversary-controlled data.

From the program's perspective, researchers have argued for defenses at a program's *attack surface* [13], which is defined by the entry points of the program accessible to adversaries because they may access adversary-controlled data. Unfortunately, programs often have a large number of library calls signifying potential entry points, and it is difficult to know which of these are accessible to adversaries using the program alone. Some experiments have estimated attack surfaces using the value of the resources behind entry points [17, 18]. However, if the goal is simply to take control of a process, any entry point may suffice. While researchers have previously identified that both the program and the system security policy may impact the attack surface definition [13], methods to compute the accessibility of entry points have not been developed.

In this paper, we compute the attack surface entry points for programs *relative to* the system's access control policy, thus overcoming the above limitations of focusing only on either one, and enabling accurate location these entry points. First, we propose an algorithm that uses the system's access control policy to automatically distinguish adversary-controlled data from trusted data based on the permissions of each program's adversaries. This constructs what we call a program's *integrity wall*[1]. We use the system's MAC (as opposed to DAC) policy for this purpose because it is immutable, thus preventing the permissions of adversaries from changing dynamically. To determine adversary access using MAC policies, past work leveraged program packages to define what is trusted by programs [31, 27]. However, the subjects associated with packages are not all necessarily trusted equally. For example, the Apache package includes user-defined CGI scripts, and clearly these cannot be trusted by the Apache webserver. Instead, we propose a novel approach for computing per-program adversaries based on the ability to modify the program's executable content.

Second, we construct a runtime analysis to collect the program entry points that access objects outside its integrity wall. Fundamental to the runtime analysis are techniques to find the program entry points (instructions in the program's binary), that receive adversary-controlled inputs. Our techniques support both binary code and several interpreted languages (e.g., Bash, PHP, Python) to enable system-wide computation of program attack surfaces. Where available, we use developer test suites for application programs; these often test multiple program configurations as well, using which we were able to associate certain entry points with configuration options that enabled them.

We evaluate a prototype runtime analysis tool on Ubuntu Linux LTS 10.04.2, using the distribution's SELinux MAC policy to build integrity walls and the application packages' test suites to guide the runtime analysis to collect attack surfaces. The tool found that this distribution's trusted computing base (TCB) processes have 2138 entry points, but only 81 attack surface entry points that an adversary could potentially exploit. While examining the system TCB attack surface, we found a previously unknown vulnerability in one entry point in a script that has been present in Ubuntu for several years. Detailed analyses of Apache and OpenSSH found an entry point in OpenSSH missed by a previous manual analysis, and demonstrates the ability of our tool to associate entry points with configuration options and find subtle, easily overlooked entry points. Also, analysis of a recent program, the Icecat web browser, revealed a previously unknown untrusted search path vulnerability, demonstrating the value in applying this analysis proactively on new programs.

[1] Adversary-controlled data lies outside the program's wall, and trusted data inside the wall.

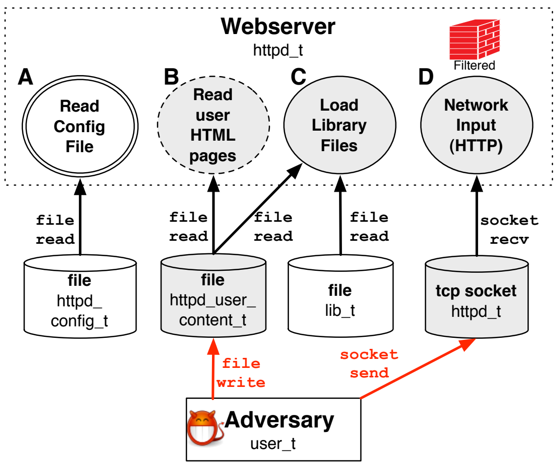


Figure 1: An example of a webserver process showing four input entry points A to D. Objects and processes are shown with their MAC labels. The shaded entry points define the program's actual attack surface (i.e., access adversary-controlled data), although only entry point D is protected by filtering. Entry point B is only activated in certain configurations, but must nonetheless be identified as part of the attack surface.

In summary, we make the following contributions:

- We propose an algorithm to construct an "integrity wall" for applications based on MAC policy and a runtime technique to precisely identify attack surface entry points in programs (including interpreted scripts) using the constructed wall,
- We present results of the attack surface for the system TCB for Ubuntu 10.04.2 and some of its applications, which helped uncover two previously unknown bugs, one present for several years in Ubuntu, showing the value of locating attack surfaces before an adversary does.

2. PROBLEM DEFINITION

The aim of this paper is to identify program entry points that access adversary-controlled objects. If an adversary can modify an object that is accessed by a program entry point that expects only safe objects, the running program can often easily be compromised.

Consider some of the entry points in a typical webserver program shown in Figure 1. During development, the application developers have realized that entry point D receives adversary input via the network, making D part of the program's attack surface. As a result, the developers have added defenses to filter input at D. The program is then deployed in a system under a particular access control policy and configuration that allows the accesses shown. Entry point A reads a configuration file, and under any reasonable MAC policy the adversary cannot access that file. Thus, A is not part of the attack surface. Suppose the administrator has enabled the *UserDir* configuration directive, allowing users to define their own HTML files (e.g., `~/public_html`). Then, entry point B receives adversary-controlled input (user-defined web pages), but the developers have overlooked this entry point because it is opened only under certain configurations and, moreover, not an obvious threat. Finally, entry point C reads in module library files (e.g., *ModCGI*) to serve a request. While this entry point is supposed to read in files labeled `lib_t` from `/usr/lib`, it has an untrusted search path bug that first searches for files in the current working directory. Hence, C exercises permissions that it is not meant to, and reads the user's `public_html` directory for libraries. An adversary can easily take control of the web server and gain its privileges if she plants a malicious module library in that directory. Thus, the adversary has found two entry points B and C into the program not anticipated by the developers.

In practice, we have seen much the same pattern. After the Apache webserver was launched in 1998, vulnerabilities were found at entry points that access log files, CGI script output, user-defined HTML files, and user-defined configuration files over a period of six years[2]. We believe that locating the attack surface proactively enables: (1) verification of where input filtering is necessary to protect the program, such as B and D, that have to handle adversary input, and (2) identification of entry points that should not be part of the attack surface, such as C, so the program or policy can be fixed. Our evaluation (Section 5) found two previously unknown vulnerabilities, one for each of the above cases.

While classical security principles stress the importance of recognizing where programs may receive adversary input (e.g., Clark-Wilson [9] requires entry points to upgrade low-integrity data), we lack systematic techniques to identify these program attack surfaces entry points. Recent work has focused on how programmers can express their attack surfaces to systems for enforcement [29, 15, 37] or for further testing [13, 17]. However, this work assumes developers already have a complete understanding of their program's attack surfaces, which experience and our results show to be incorrect. Our results demonstrate that both mature and new programs may have undefended attack surface entry points, and many entry points are accessible to adversaries in subtle ways.

Assumptions. Our work calculates attack surface entry points in programs and not the attack surface of the OS kernel itself. Thus, we assume the OS kernel to be free from vulnerabilities that a local attacker can exploit. Further, we assume that the reference monitor enforcing access control in the OS enforces a MAC policy, and satisfies *complete mediation* and is *tamperproof* [2]. This implies that the only way for local adversaries to attack programs is through rules specified in the OS MAC policy.

3. DESIGN

Calculating the attack surface has two steps. First, for a particular subject (e.g., `httpd_t`), we need to define its adversaries (e.g., `user_t`), and locate OS objects under adversarial control (e.g., `httpd_user_content_t`). We do this using the system's MAC policy. Next, we need to identify the program entry points (e.g., entry points B, C, D) that access these adversary-controlled objects. Statically analyzing the program cannot tell which permissions are exercised and which OS objects accessed at each entry point, and thus we use a runtime analysis to locate such entry points. In this section, we detail solutions to these two steps.

[2]CVEs 1999-1206, 2001-1556, 2002-1850, 2004-0940, 2004-2343 respectively

3.1 Building Integrity Walls

A program may receive many inputs. However, not every input into a program is necessarily under the control of adversaries. A program *depends on* (i.e., trusts) some inputs (e.g., `etc_t` and `lib_t` in Figure 1), whereas it needs to *filter* (i.e., protect itself from) other inputs (e.g., `httpd_user_content_t` in Figure 1). Our insight is that the system's MAC policy enables differentiation between those OS objects that a subject depends on and those OS objects that it needs to filter. This is simply because a properly designed MAC policy limits the modification of OS objects that a particular subject s depends on only to subjects that are trusted by s, and any other object is untrusted and needs to be filtered on input. Thus, if we identify the set of subjects trusted by s, we can then derive the trusted and untrusted objects for s from the MAC policy.

Integrity Wall Approach. The observations that we use to calculate the set of trusted subjects are outlined below. First, a process fundamentally depends on the integrity of its executable program file. Thus, a subject in the MAC policy has to trust other subjects that have the permission to modify its executable program file[3], called *executable writers*. While we could expand this definition to include all code used by a process, such as libraries, we find that the set of labels for approved libraries are ambiguous and that these are covered by the other cases below.

Second, a process depends on the integrity of its underlying system. If the kernel can be compromised, then this process can be trivially compromised. Thus, all subjects depend on the subject labels with permission to modify any kernel objects, called *kernel subjects*. Naturally, each subject also depends on the executable writers of the kernel subjects as well. This combination forms the system's *trusted computing base* (TCB).

Third, several applications consist of multiple distinct processes, some of which are trusted and some not. For example, `htpasswd` is a helper program for Apache that maintains the password file `.htpasswd`. Intuitively, Apache depends on this program to maintain the password file properly. On the other hand, Apache should filter inputs from user-defined CGI scripts. We state that a subject label s depends upon a *helper subject*, if: (1) the two subject labels are part of the same application (e.g., package) and (2) the helper subject's executable writers are in the application or trusted by s. Identifying that two subject labels are part of the same application is often easy because MAC policies are now written per application (e.g., there is an SELinux policy module for Apache).

Integrity Wall Algorithm. The problem is thus to compute for each subject a partition of the set of MAC policy labels P based on whether the subject depends on the label or not, based on the three criteria above, forming that subject's integrity wall. An integrity wall for a subject s is a partition of the set of labels[4] in the system policy P into sets I_s and O_s, such that s depends on labels in I_s ("inside

[3]Typically, MAC policies are designed by assigning permissions to each executable independently, which means there is often a one-to-one mapping between subject labels and executable files.

[4]The set of object labels includes the set of subjects labels in the policy, but not vice versa. Also, we use the terms subject and object for subject label and object label from this point forward when unambiguous.

the wall"), and filters inputs from labels in O_s ("outside the wall").

The integrity wall derivation computes $I_s = P - O_s$ from MAC policy containing relations x WRITE y and x WRITEX y, which mean subjects of label x can write objects of label y and subjects of label x can write executable file objects of subject y, respectively.

1. The *kernel subjects* $K \subseteq P$ of a system are:
   $K = \{s1 \mid \exists o \in \text{KERNEL}(P)$, where $(s1, o) \in \text{WRITE}\}$
2. The *trusted computing base* $T \subseteq P$ of a system is:
   $T^0 = K$;
   $T^i = T^{i-1} \cup \{s2 \mid \exists s1 \in T^{i-1}, (s2, s1) \in \text{WRITEX}\}$;
   $T = \bigcup_{i \in N} T^i$
3. The *executable writers* $E_s \subseteq P$ for a subject s are:
   $E_s^0 = s$;
   $E_s^i = E_s^{i-1} \cup \{s2 \mid \exists s1 \in E_s^{i-1}, (s2, s1) \in \text{WRITEX}\}$;
   $E_s = \bigcup_{i \in N} E_s^i$
4. The *helper subjects* $H_s \subseteq \text{APP}(s)$ for a subject s are:
   $H_s = \{s1 \mid (s1 \in (\text{APP}(s) - \{s\})) \wedge (E_{s1} \subseteq (\text{APP}(s) \cup E_s))\}$
5. The *trusted subjects* $T_s \subseteq P$ for a subject s are:
   $T_s = T \cup E_s \cup H_s$
6. The *trusted objects* $I_s \subseteq P$ for a subject s are:
   $I_s = T_s \cup \{o \mid \nexists s1 \in (P - T_s), (s1, o) \in \text{WRITE}\}$

First, we compute the kernel subjects (i.e., subjects with WRITE access to KERNEL(P) objects) and TCB for the system at large. The TCB is derived from a transitive closure of the writers of the kernel subjects' executables (WRITEX). Then, for each subject we compute its executable writers (again, using transitive closure) and its helper subjects. Helper subjects must be part of the same application (APP(s)) as the target subject s and can only be modified by a subject outside the application that is trusted by s. Thus, `htpasswd` is an Apache helper, but user scripts are not, as their executable is written to by an untrusted subject (the user). Finally, we collect the trusted objects for the subject: the set of objects that are only modified by the trusted subjects. A problem is that some objects are written only by trusted subjects, but are known to contain adversary-controlled data, such as log files. We assume these objects to be untrusted. More such cases are discussed in the evaluation.

This method computes the object labels inside the integrity wall for a subject label, and all other objects labels are outside the integrity wall for that subject. Access to objects outside the wall will be the focus in building each program's attack surface.

3.2 Identifying Attack Surfaces

Using an integrity wall for a subject, we can find the attack surfaces of all programs that run under that subject. As noted, it is impractical to identify these entry points statically, because any system call is authorized to access any object to which the program's subject is authorized. Therefore, we propose a runtime analysis to locate entry points. Runtime analysis provides a lower-bound for the number of entry points in an attack surface, but nonetheless, we have found many non-trivial attack surfaces with recent vulnerabilities and we identified new vulnerabilites (Section 5).

The most important design decision is to define what an entry point is. To find the program entry points, we obtain the process's execution stack at the time of the system call. Consider a program performing a system call that receives input, through a stack of function calls $F_1, F_2, \ldots, F_n$, where F_i calls F_{i+1}. The entry point into the program occurs at the greatest index i where F_i is not trusted to filter all input. That is, we may trust `libc` to protect itself from untrusted input, making the caller of `libc` the entry point. This is often, but not always, the program executable instruction that invoked the library call.

Developing a runtime analysis for identifying program attack surfaces must meet the following requirements.

1. All security-sensitive operations from all processes must be mediated,
2. The subject and object labels of each operation are available, and
3. The context of the process execution (e.g., the instruction pointer and process stack) is available.

First, both user-level and kernel-level mechanisms have been designed to mediate system calls, but the use of kernel-level mediation is preferred in this case because: (1) multiple security-sensitive operations and objects may be accessed in one system call, and it requires significant parsing effort in user-space to find them all accurately and (2) all processes can be mediated in a single location for low overhead. In several modern operating systems, reference monitors have been implemented to mediate all security-sensitive operations [36, 35, 32], which we extend to detect accesses of objects outside the integrity wall.

Second, we need to know the subject and object labels of the operation to determine if this operation is outside the integrity wall for that subject. The label information is obtained from the reference monitor module enforcing system security (e.g., SELinux [22] and AppArmor [21] for Linux). We use this information to determine whether the subject is accessing an object outside its wall based on the trusted objects I_s or its set complement O_s.

Third, when an untrusted object is accessed, we find the process' user stack at the time of the call to find the entry point. The main challenge is to search up the stack to find the first stack frame that is not trusted to filter all inputs. Each frame must be mapped to its code file to determine whether it is trusted for all inputs. We use the virtual memory mappings to determine the code file, and we maintain a table of those that are fully trusted. The specific mechanism is described in the implementation.

Finally, we log entry points to user-space, so they can be collected and analyzed. A log entry record consists of the subject and object labels of the operation and the entry point in the process' user stack at the time of the operation.

4. IMPLEMENTATION

In this section, we describe how we implemented our design on Ubuntu 10.04.2 LTS Desktop Edition running a Linux 2.6.35 kernel, with SELinux as the MAC enforcement mechanism. We first describe our implementation to construct the integrity wall for subjects, and then our modification of the Linux kernel to log the entry points at runtime. We also explain how we extended our system to deal with interpreters. Our modifications added 1189 lines of code to the Linux 2.6.35 kernel: 588 lines of code for interpreter processing and the rest to fetch the stack backtrace, detect if the operation is untrusted, and log fresh entries to userspace.

4.1 Integrity Wall Construction

We implement the design described in Section 3.1. We implement the algorithms for each step in XSB/Prolog. In total, the algorithms required 101 Prolog statements, 77 of these were for parsing the SELinux policy, and the rest for the wall generation. The main input is the system's SELinux MAC policy, which consists of a set of policy modules for individual Linux applications deployed on the system[5]. We describe implementation of the algorithms in terms of TCB computation and the subject label's integrity wall computation.

To construct the TCB of the system, we manually identify a set of 13 kernel objects ($\textsc{Kernel}(P)$), write access to which could directly compromise the kernel (e.g., `/dev/kmem`). Using Steps 1 and 2 of Section 3.1, the SELinux policy and the kernel objects are used to identify the set of subjects that can write to these kernel objects and to perform a transitive closure of the writers of the binaries of the kernel subjects. The SELinux policy identifies all write operations ($\textsc{Write}$) and the objects that may be executables for a subject for computing $\textsc{Writex}$. SELinux defines the object types[6] for initiating a subject using `type_transition` rules. The object type in such rules corresponds to the label of the corresponding executable file.

Using the SELinux MAC policy, we compute the integrity wall for SELinux subject types in the Ubuntu distribution, using Steps 3-6 in Section 3.1. To identify helper subjects, we need to identify the subjects that are part of the same application ($\textsc{App}(s)$). SELinux offers policy modules for applications, and we consider all subjects defined in an application policy module as being part of the same application. All TCB subjects use the same integrity wall.

As a special case, we force all log files to be outside the integrity wall of all subjects. Log file types are easily identified in the SELinux policy (e.g., `var_log_t`).

4.2 Identifying Attack Surfaces

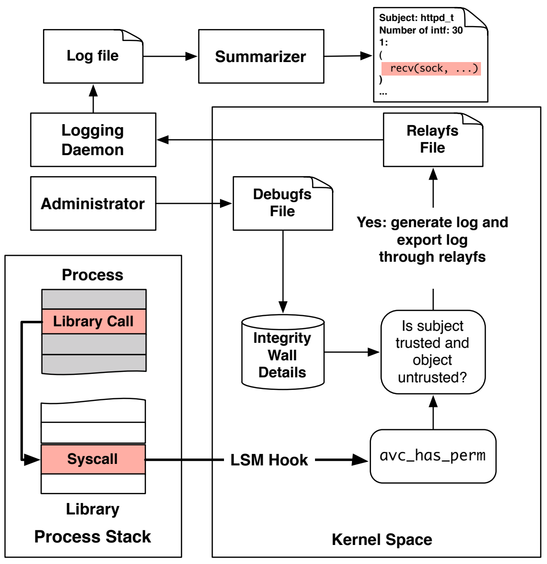
Once we have the integrity wall, we use it to locate operations crossing the wall using runtime analysis. Figure 2 details our implementation in Linux, which leverages the Linux Security Modules (LSM) framework [36]. We use the SELinux LSM running in the Linux kernel to find untrusted operations. This satisfies the three requirements for identifying untrusted operations (Section 3.2). First, it mediates all security-sensitive operations using the LSM interface. Second, we instrument the SELinux access decision function, `avc_has_perm`, which authorizes a subject type to perform an operation on an object of a particular object type. Finally, the kernel always has details about the currently executing process.

Our implementation enables: (1) uploading of integrity walls; (2) identifying operations outside the wall for a process; (3) finding the process entry point; and (4) logging the operation. We examine implementation of these below.

To upload the integrity walls into the kernel, we export

[5]Many, but not all Linux applications have SELinux modules. Those programs without their own modules run under a generic label, such as `user_t` for generic user programs.

[6]In SELinux, labels are called *types*.



```
Subject: httpd_t
Number of intf: 30
1:
(
  recv(sock, ...)
)
...
```

Figure 2: When the program makes a syscall, the SELinux function `avc_has_perm` is hooked through LSM hooks. If the calling function reads an object with a label outside the wall for that process, then a log entry including the entry point into the user program is generated to a relayfs file. A userspace logging daemon collects this data into a log file, and a summarizer analyzes this file to output the attack surface. The integrity walls for each subject label are pushed into the kernel via a debugfs file.

a `debugfs` file to communicate the integrity wall for each subject type to the kernel.

To identify operations accessing objects outside a subject's wall, we look for operations that input data (i.e., read-like operations). We use the permission map from Apol [33], which identifies whether an operation is read-like, write-like, or both. Interesting to note is that the permission map classifies operations from most covert (least actual input, such as file locking) to most overt (most actual input, such as file reading). We ignore covert operations, excepting for directory searches. We find these valuable because they indicate the presence of an attack surface if the directory is untrusted, even if the file itself is not present. For example, when a program searches for libraries in the current working directory (untrusted search path), the library file itself is not present in benign conditions, but the search indicates an attack surface.

Once we identify an operation that reads from outside the wall, we find the entry point into the process. We first obtain the user stack trace of the process, and then identify the exact entry point. The user stack trace is available by a simple unrolling of the linked list of base pointers on the userspace stack. Such functionality is already available using the `ftrace` framework in the Linux kernel, which we use.

To find the entry point, we search for the stack frame that belongs to a code object that cannot protect itself from all input. In Linux, the `vma_struct` is associated with the name of its code object file (`current->comm`). Thus, for an instruction pointer, we retrieve its code object file through the `vma_struct` to identify entry points. Due to address-space randomization, the exact IP may vary across different runs of the same process. Thus, we use the offset from the base of the binary, which is a constant. Then, this informa-

```
PROCESS: httpd  CONTEXT: system_u:system_r:httpd_t
Number of entry points: 30
1:
(
    0x2EC4A, /home/user/httpd-2.2.14/server/core_filters.c:383,
      15, FILE__READ, system_u:object_r:httpd_user_content_t
)
-------------------------
2:
(
    0x6758A, /home/user/httpd-2.2.14/server/listen.c:140,
      38, TCP_SOCKET__LISTEN, system_u:object_r:httpd_t
)
```

Figure 3: A log entry from recording of untrusted operations. Two entry points for Apache, with its location information.

tion can be used offline to find the exact line of code in the program if versions of the binaries are available with debug information (many of these are readily available in Ubuntu repositories).

To export the data to userspace, we use `relayfs`. A userspace daemon reads the kernel output and dumps the output to a file. The daemon registers itself with the kernel so it itself will not be traced. For each untrusted operation, we log the following: (1) process name; (2) process ID; (3) the entry point IP into the process as an offset from the base of the binary; (4) the SELinux context of the subject[7]; (5) the SELinux context of the object; (6) operations requested; and (7) filename, if object is a file.

Once the log is available in userspace, it is parsed to output a list indexed by process name(Figure 3). For each process, each entry point indexed by IP is listed, with the operation(s), types of data read through that entry point, and the number of times that entry point is invoked. If debug information is available for the process, we also print the C source line of code that the entry point is associated with.

4.3 Finding Attack Surfaces in Interpreted Code

For interpreted programs, a normal backtrace of the user stack will supply an entry point into the interpreter, and not the script that it is executing. Several shell scripts are run during normal operation of the system, some of which have fallen victim to adversaries, so we also have to accurately identify the attack surfaces of interpreted programs.

We built a kernel-based mechanism for extracting entry points from programs in a variety of interpreted languages (PHP, Python, Bash). This mechanism takes advantage of the common architecture of interpreters. First, these interpreters execute their programs by running each language instruction in a function that we call the *fundamental loop function*. Each interpreter also maintains a global *current interpreter object* that represents the state execution of the program, much like a process control block describes processes. Second, when an error occurs, the fundamental loop functions each call *print backtrace* function, that extracts the stack frames of the currently executing program from the current interpreter object.

To collect entry points for interpreted code, we made interpreter state visible to our kernel mechanism. This involved creating kernel modules that are aware of each interpreter: (1) obtaining access to each's current interpreter object from their ELF binary symbol tables and (2) using each's print backtrace functions to find entry points. First, the ELF binary loader already contains mechanisms for accessing the symbol table during program loading that we used to gain access to the desired references. Second, we integrate the backtrace code from the interpreter into the kernel module to find entry points. This task is complicated because we need to use this code to access user addresses from kernel space. To do this safely, we use macros to handle page faults that may result from user space access (`copy_from_user`) and remove code that causes side-effects (any writes to user space). Ultimately, very little code needed to be transferred from user space: Bash required 59 lines of code, most of it to handle hash tables in which it stores its variables, whereas PHP required just 11 lines of code. Ultimately, 588 lines of code were added to the kernel for the three interpreters, but 391 lines are for defining data structure headers.

[7]An SELinux context includes a type and other information, including a user identity and role. We are mainly interested in the type.

4.4 Enforcing Attack Surfaces

We note that the same infrastructure that logs attack surface entry points can also enforce them. In other words, any access crossing the integrity wall would be blocked unless made through one of the authorized entry points for that program and between appropriate types. When any previously unknown entry point crossing the integrity wall is found, its details can be reported to the OS distributor much like crash reports are sent currently, who can decide if the entry point is valid. Note that our enforcing mode can block entry points exercising improper permissions such as untrusted search paths (even those having previously unknown bugs), while access control cannot (as the process might legitimately have those permissions at another entry point). To make our tool performant for online logging and enforcement, we made some enhancements. The integrity walls for subjects are stored as a hash table so looking up whether an object is inside or outside the wall is fast. Also, we only log operations if they have not been logged already.

5. EVALUATION

In this section, we present the results of our analysis of attack surfaces for the system TCB of an out-of-the-box install of Ubuntu Desktop LTS 10.04.2, with the SELinux policy from the repositories. Our aim in this evaluation is to demonstrate the effectiveness of our approach in computing the attack surfaces for all the system programs in a widely-used Linux distribution in relation to its default SELinux policy. In addition, we performed a detailed study of application programs, including Apache **`httpd`**, **`sshd`** and the Icecat browser, the GNU version of Firefox. While the Apache and OpenSSH are mature programs, we show that our approach can identify attack surface entry points that are easily overlooked. Icecat is a relatively new program, so its analysis demonstrates how our approach may aid in the proactive defenses of immature programs.

We found the following results. For the system TCB, we found that: (1) our analysis was able to obtain an attack surface of 81 entry points, including in scripts and some subtle entry points, 35 of which have had previous vulnerabilities, and (2) the attack surface of these programs is a small percentage of their total number of entry points. Examining the system TCB attack surface, we found a vulnerability in one entry point in a script that has been present in Ubuntu for several years. For Apache and **`sshd`**, we were able to associate attack-surface entry points with the configuration option that enabled them by correlation with the configura-

	Types Inside Wall		Types Outside Wall	
	Subjects	Objects	Subjects	Objects
System TCB	111	679	153	142
Apache (httpd_t)	118	700	146	121

Table 1: Wall statistics for the system TCB types and Apache. Subject types correspond to processes, and object types correspond to OS objects (e.g., files) and processes.

tion used by their test suites. In `sshd`, we found an entry point in the privilege-separated part that was missed by earlier manual identification [29]. In Icecat, we found an entry point that was part of the attack surface due to a bug.

5.1 Policy Analysis

This section presents the results of the wall generation algorithm described in Section 3.1, on Ubuntu 10.04.2's SELinux policy. This policy used 65 application policy modules, and had 1058 types (subject and object) in total.

System TCB. We first need to locate the TCB that is common to all applications. We build the TCB as described in Section 4.1. In total, we had 111 subject types in the TCB.

Wall Generation Results. Table 1 shows the number of subject types inside and outside the wall for the system TCB subject types and the Apache subject types and the resulting number of high and low integrity object types. We note that only seven new subject types are added to the integrity wall for Apache over the system TCB subject types, which it must already trust. Interestingly, the number of high integrity object types given this wall outnumbers low integrity types by more than 4:1.

We confirmed that for the system TCB programs we examined, the integrity wall derived from the policy corresponded to our intuitive notion of dependence and filtering, (i.e.,) configuration files and library files were within the wall, and user-controlled input outside; we present more details of the wall when we discuss individual applications.

Violating permissions. We found that of 115,611 rules in our SELinux policy, 34.4% of these rules (39,848) crossed across the system TCB integrity wall, allowing input from object types outside the system TCB to subjects in the system TCB. The attack surface will consist of entry points in TCB programs that exercise a permission that crosses the wall. These cannot be found from the MAC policy; we need information from the program.

5.2 Runtime Analysis

5.2.1 System TCB

Evaluation of the system's TCB demonstrates that: (1) the number of attack surface entry points is a small percentage of the total number of entry points, (2) some attack surface entry points are subtle, and (3) even for mature programs in the system TCB, it is beneficial to locate the attack surface, as demonstrated by a bug we found in an entry point in a mature script that sets up the X server. We gathered the attack surface over several days on a system in normal use, involving bootup, typical application usage, and shutdown.

We found that only 13.8% (295 of 2138) of the total entry points are part of the attack surface (Table 2). For example, we found many entry points accessing trusted objects such as `etc_t`; these entry points would not be part of the attack surface. Thus, simply listing all possible entry points in TCB programs as part of the attack surface would be a significant overapproximation, and not very useful for analysis. Of the 295 attack surface entry points across various programs in the TCB, that received untrusted input, 81 are overt operations (Section 4.2); 35 of these have had input filtering problems, many recently discovered for the first time. Five Bash scripts add a total of 8 entry points to the attack surface (Table 3). In addition, we found a previously unknown vulnerability at an entry point in a script that sets up the X server that has been around for several years, which we discuss below. This leads us to believe that identification and examination of such entry points prior to deployment is key to preventing exploits.

Runtime analysis in inherently incomplete. To examine completeness, we ran our kernel module in an enforcing mode (Section 4.4), where any access crossing the system TCB integrity wall was blocked unless made through one of the entry points in Table 2 and between appropriate types. We did not note any new accesses, and since we have a conservative adversary model (including unprivileged users), we believe our set of entry points to be complete for a default Ubuntu 10.04.2 Desktop distribution in relation to its SELinux policy.

We located various subtle entry points that are part of the attack surface. We illustrate this using the example of `logrotate`. `logrotate` has an entry point that reads from `user_home_t`, and the source code for this entry point called a library function that gave little hint as to why this was happening. The reading is actually done inside a library function in `libpopt` attempting to read its configuration file. As another example, we found entry points calling `libc glob()`. This function performs the system call `getdents` returning untrusted directory filenames. A recent untrusted filename attack on `logrotate` (CVE-2011-1155), was found at this entry point. Neither of the above entry points are as a result of simply calling `read()` in the source code, and can be easily missed by manual code inspection.

We examined some of the entry points identified, to see if we could locate any obvious problems. The script corresponding to entry point 2 in Table 3 is responsible for setting up the `/tmp/.X11-unix` directory, in which the X server creates a Unix-domain socket that clients can connect to. This flow is into `initrc_t` from `tmp_t` (Table 2). However, we found that it is vulnerable to a time-of-check-to-time-of-use (TOCTTOU) vulnerability. Looking at the script makes it fairly clear that the developer did not expect a threat at this entry point. This script has existed as part of Ubuntu distributions since at least 2006, and is an example of how locating the attack surface made the problem obvious. We believe that a more thorough testing of the entry points uncovered may expose further vulnerabilities; however, that is outside the scope of this paper.

5.2.2 Apache Webserver

We use our tool to evaluate a typical webserver deployment, the Apache webserver (version 2.2.14) with mod_perl. For the wall generation, of particular interest are object types in the SELinux policy module for Apache that were not included in the application TCB, four `httpd_user` types and `httpd_log_t`. For the runtime analysis for Apache, we ran the `Apache::Test` perl module, which contains test cases

TCB Type	**Total Entry**	**Viol. Entry**	**Program**	**Overt Violating Entry**	**Object Type Accessed**	**Bug ID / Notes**
apmd_t	3	3	acpid	1 Unix socket	apmd_t	CVE-2009-0798
avahi_t	38	14	avahi-daemon	3 * Unix socket	system_dbus_daemon	CVE-2007-3372
				1 Netlink socket	avahi_t	CVE-2006-5461
				1 UDP socket read	avahi_t	CVE-2006-6870
consolekit_t	37	3	console-kit-daemon	1 file	tty_device_t	–
				1 Unix socket	system_dbus_daemon	CVE-2010-4664
				1 file	consolekit_log_t	–
cupsd_t	56	10	cupsd	1 TCP socket	cupsd_t	CVE-2000-0540
				1 file	print_spool_t	–
devicekit_disk_t	72	6	udisks-daemon	1 * 4 unix socket	system_dbus_daemon	CVE-2010-0746
				1 netlink socket	devicekit_power_t	–
devicekit_power_t	97	7	upowerd	1 * 2 unix socket	devicekit_power_t	–
				1 netlink socket	devicekit_power_t	–
dhcpc_t	15	2	dhclient3	1 raw socket read	dhcpc_t	CVE-2009-0692
			nm-dhcp-client.action	1 unix socket	system_dbus_daemon	–
getty_t	18	3	getty	1 file read	initrc_var_run_t	–
hald_t	188	28	hald	1 unix socket	system_dbus_daemon	–
			hald-probe-serial	1 file	tty_device_t	–
			hald-addon-storage	1 unix socket	system_dbus_daemon	–
			hald-addon-acpi	1 unix socket	apmd_t	Ubuntu Bug 230110
initrc_t	479	23	sh	1 file read	initrc_var_run_t	–
			sh	2 * dir read	tmp_t	**Prev. unknown**
			telinit	1 * 2 file read	initrc_var_run_t	–
init_t	319	27	plymouth	1 file read	devpts_t	–
			ureadahead	1 file read	user_home_t	–
			init	1 unix socket	system_dbus_daemon	–
			sh	2 file read	tmp_t	–
			loadkeys	1 file read	devpts_t	–
local_login_t	152	10	login	1 * 2 file read	initrc_var_run_t	CVE-2008-5394
				1 unix socket	system_dbus_daemon	–
				1 file read	user_home_t	(motd) CVE-2010-0832
				1 dir search	user_home_dir_t	(hushlogin)
				1 dir search	user_home_dir_t	CVE-2010-4708
			python	1 dir search	user_home_dir_t	Python search path
logrotate_t	41	6	logrotate	1 file read	generic - log files	–
				1 dir search	user_home_dir_t	(libpopt)
				1 dir read	var_log_t	CVE-2011-1155
NetworkManager_t	76	45	NetworkManager	1 netlink socket	NetworkManager_t	–
				1 unix socket	system_dbus_daemon	CVE-2009-0578
			sh	2 * dir search	tmp_t	–
ntpd_t	24	4	ntpdate	1 udp socket	ntpd_t	CVE-2001-0414
restorecond_t	17	9	restorecond	1 * 3 file read	generic - all types	–
				1 dir read	user_home_dir_t	–
rtkit_daemon_t	20	9	rtkit-daemon	1 unix socket	system_dbus_daemon	–
sshd_t	78	11	(Discussed	(5 in privileged	–	2 vulns
			in Table 5)	part)	–	–
syslogd_t	29	1	rsyslogd	1 udp socket	syslogd_t	CVE-2008-5617
system_dbusd_t	63	15	dbus-daemon	1 * 3 unix socket	system_dbus_daemon	CVE-2008-3834
udev_t	217	25	udevd	1 * 2 netlink socket read	udevd_t	CVE-2009-1185
			sh	1 file read	tty_device_t	–
xdm_t	56	16	gdm-binary	1 file read	user_home_t	CVE-2006-1057
			gdm-simple-slave	1 unix socket	system_dbus_daemon	–
				1 file read	initrc_var_run_t	–
				1 file read	xdm_tmp_t	–
			gdm-session-worker	1 file read	xauth_home_t	CVE-2006-5214
				1 dir search	user_home_dir_t	CVE-2010-4708
xserver_t	43	18	Xorg	1 * 3 file read	xdm_tmp_t	–
				1 unix socket	xserver_t	CVE-2007-1003
				1 netlink socket	xserver_t	–
				1 shared memory	unconfined_t	CVE-2008-1379
Total	**2138**	**295**		**81**		**35**

Table 2: Attack surface for the system TCB. The first column is the TCB type we consider, the second the total number of entry points for all programs running under that type, and the third the number of violating entry points that cross the integrity wall. Next, we list the specific binary with its overt violating entry points (Section 4.2) and the object type accessed that causes the entry point to be violating. We also identify vulnerabilities caused due to insufficient filtering at the overt entry points (we could not find any for the covert entry points). When multiple vulnerabilities are available for an entry point, the chronologically earliest is listed. Highlighted rows are discussed further in text.

ID	Source Script:Line Number	Source Type	Target Type
1	/lib/udev/console-setup-tty:76	udev_t	tty_device_t
2	/etc/rcS.d/S70x11-common:33,47	initrc_t	tmp_t
3	/usr/lib/pm-utils/functions:30	initrc_t	initrc_var_run_t
4	/etc/NetM/dispatcher.d/01ifupdown:27,29	NetworkManager_t	tmp_t
5	/etc/init/mounted-tmp.conf:44,45	init_t	tmp_t

Table 3: Entry points in Bash scripts.

ID	Source File:Line Number	Object Type Accessed	Description	Config Option
1	server/util.c:879	httpd_user_htaccess_t	read user .htaccess file	AccessFileName
2	server/core_filters.c:155	httpd_t	read tcp socket	-
3	server/core_filters.c:383	httpd_user_content_t	read user HTML file	UserDir
4	server/connection.c:153	httpd_t	read remaining tcp data	-
5	os/unix/unixd.c:410	httpd_user_script_exec_t	execute CGI user script	Script

Table 4: Apache entry points receiving low-integrity data

generated by the Apache developers. We found 30 entry points for the Apache webserver, of which 5 received untrusted operations. Details are in Table 4.

We located several entry points accessible to adversaries. Network attacks being well understood, we list implications of the entry points accessing local untrusted data (1, 3 and 5 in Table 4). `httpd_user_htaccess_t` denotes the user-defined configuration file `.htaccess`. Previous problems with this entry point are Bugtraq IDs 8911, 11182, 15177. `httpd_user_content_t` are user-defined web pages that Apache serves. A vulnerability due to incorrect parsing of the HTML files is BID 11471. Entry point 5 is where Apache forks a child to execute a user-defined CGI script - the `exec` operation reads an untrusted executable is untrusted (BID 8275), and could easily be missed by manual analysis.

As mentioned before, our tool was able to associate some entry points with the configuration option that controlled the entry point; different application configurations may expose different attack surfaces. This knowledge is helpful to administrators, who can view the effect of their configuration on the attack surface.

5.2.3 *Secure Shell Daemon*

We also performed a study on the SSH daemon, `sshd` (v. 5.1p1). In total, there were 78 entry points, of which 27 required filtering. 14 of these which correspond to overt input are listed in Table 5. Entry points 12, 13 and 14 were opened by non-default configuration options. Of key interest, is that OpenSSH has been re-engineered to separate the privileged operations from those that are unprivileged to prevent vulnerabilities [25]. This work focuses on two attack surface entry points that communicated data from the unprivileged SSH daemon process to the privileged, master SSH daemon process. Using our tool, we found another entry point (7) that reads the `authorized_keys` file in the `.ssh/` directory of users. Since this is modifiable by users, it could be of low integrity [28], and our wall indicates this. This entry point was also missed in a manual analysis to configure SELinux policies to enforce privilege separation [29], showing the importance of an automated technique like ours. Entry points 13, 14 may present similar issues for those configurations.

5.2.4 *Icecat*

We also performed a study on the GNU version of Firefox, Icecat. The objective of this study was to look at a relatively less-known project, to see if we could find any problems using our tool. We envision this to be a typical use-case of our tool. In total, we found 18 entry points for Icecat, of which 4 accessed untrusted data. On closer examination of the attack surface, we found an entry point that searched the directory `user_home_dir_t`, whose code was in the dynamic loader/linker library `ld.so`. We suspected an untrusted library search path, and confirmed that this indeed was the case. This could easily be exploited by an adversary-controlled library that the user downloads to her home directory. The developers accepted our patch [5].

5.3 Performance

Micro- and macro-benchmarks showed acceptable performance overheads for online logging and enforcement. For example, `stat` system call takes an unmodified kernel took 8.5μs on average. Overhead for checking if the subject was trusted and the object was untrusted took an additional 0.2μs. If the access was untrusted, the logging mode added an overhead of 1μs, whereas the enforcement mode added an overhead of 0.1μs. The `sshd` test suite ran in 318.29s on the unmodified kernel, whereas configured with an integrity wall for `sshd` with both enforcement and logging enabled took 318.81s.

6. RELATED WORK

Taint tracking has been used to track the flow of untrusted input to a program, and find places where it may affect the integrity of the program. Tracking can be done for whole systems [4, 8] or for specific processes [19, 26]. However, these systems expect manual specifications of taint entry points. For example, [19] considers data “originating from or arithmetically derived from untrusted sources such as the network as tainted”. However, it is not clear that all entry points that receive low integrity data are locatable manually. Our tool provides this origin input to taint tracking systems.

Manadhata et al. [17] calculate an attack surface metric for programs based on methods, channels and data. They prepare a list of input and output library calls from `libc` that are used to determine the methods. Although this is

ID	Source File:Line Number	Object Type Accessed	Description	Config Option
1*	monitor_wrap.c: 123	sshd_t	Master-slave Unix socket read	UsePrivilegeSeparation
2	msg.c:72	sshd_t	Unix socket read	-
3	msg.c:84	sshd_t	Unix socket read	-
4	sshd.c:442	sshd_t	TCP socket read	-
5	dispatch.c:92	sshd_t	TCP socket read	-
6	packet.c:1005	sshd_t	TCP socket read	-
7*	misc.c:627	user_home_ssh_t	∼/.ssh/.authorized_keys file read	AuthorizedKeysFile
8*	channels.c:1496	ptmx_t	pseudo-terminal read	-
10	serverloop.c:380	sshd_t	fifo file read	-
11	loginrec.c:1423	initrc_var_run_t	read utmp	-
12	session.c:1001	user_home_ssh_t	∼/.ssh/.environment file read	PermitUserEnvironment
13*	hostfile.c:222	user_home_ssh_t	∼/.ssh/known_hosts file read	IgnoreUserKnownHosts
14*	auth-rhosts.c:82	user_home_ssh_t	∼/.ssh/.rhosts file read	IgnoreRhosts

Table 5: `sshd` entry points that may receive low-integrity data. Entry points marked with * are in the master part of the privilege-separated daemon.

useful for a first approximation, it does not distinguish between entry points receiving high-integrity input and those receiving low-integrity input. In our analysis, only a small percentage (13.8%) of the entry points were found to receive data of low-integrity. Hence, a simple listing of all such library methods may not give a true picture of work required to secure an application. Further, a library may be called through several layers of libraries, and the context of a lower-layer library call may not be relevant through several layers. We identify the point in the application that receives low-integrity input, which is more helpful to application developers than a low-level library function that may be called in different contexts.

Several practical integrity models [16, 31, 29, 15] are related to our work. UMIP [16] and PPI [31] identify trusted subjects that need to maintain their integrity on receiving low-integrity input. Though their goals differ from ours, they also build integrity walls. UMIP builds integrity walls system-wide based on the DAC policy, whereas PPI uses package dependencies for the same. However, they identify trusted processes as a whole and do not identify entry points within a process, which we have seen to be necessary. Further, they only consider system-wide integrity walls, and not per-application. Flume [15] allows entry point-level control, but leaves the specification of the policy up to the user, who has to decide which entry points to allow to receive untrusted input. Such policy could benefit from knowledge of the attack surface. Shankar et. al [29] identify that we need to verify input filtering for entry points that receive low-integrity input. However, they identify entry points manually, and missed an entry point in `sshd` that we identified using our automated approach.

Bouncer is a system that uses knowledge of vulnerabilities to generate filters automatically [10]. It symbolically executes the vulnerable program to build a filter that covers the particular attack and generalizes the filter to cover other unauthorized inputs without preventing legitimate function. EXE automatically generates inputs that will crash a program [6]. Both of these systems would benefit from knowledge of the attack surface of a program. In the latter case, this will focus use on legitimate entry points for consideration. The inputs that cause failure may then be used to generate filters via Bouncer.

We could also have leveraged system call interposition [24, 12, 1, 3, 11] to monitor objects accessed by a program, instead of doing it in the kernel. However, as noted in Section 3.2, we have to maintain a list of system calls that causes inputs, know the sets of objects accessed by each of these calls, and fetch the security contexts of these objects from the kernel – all duplicating information readily available in the kernel. Also, system call interposition has high overhead and is challenging to do system-wide.

7. CONCLUSION

In this paper, we introduced an approach to identify attack surfaces in programs with respect to an integrity wall constructed from the system's security policy. We implemented a system in the Linux kernel that enabled precise identification of attack surface entry points, even in interpreter scripts. Our results indicate that accurate location of attack surfaces requires considering a program in relation to the system's access control policy. For the system TCB in an Ubuntu 10.04.2 Desktop system, we obtained an attack surface of 81 entry points, some subtle; 35 of these have had past vulnerabilities, many recently. Our attack surface indicated an entry point in `sshd` that was missed by earlier manual analysis, and an entry point in the GNU Icecat browser that was due to an untrusted search path bug. Further, our attack surface helped us find a bug in an entry point of the system TCB of Ubuntu that has been around for several years. We envision that our tool will be used on new programs to identify attack surfaces before an adversary does and prepare defenses, moving us away from the current penetrate-and-patch paradigm.

8. REFERENCES

[1] A. Acharya *et al.* MAPbox: Using parameterized behavior classes to confine untrusted applications. In *USENIX Security*, 2000.

[2] J. P. Anderson. Computer Security Technology Planning Study, Volume II. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronics Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, October 1972.

[3] A. Berman *et al.* TRON: Process-specific file protection for the UNIX operating system. In *USENIX TC '95*, 1995.

[4] E. Bertino *et al.* A system to specify and manage multipolicy access control models. In *POLICY*, 2002.

[5] run-icecat.sh possible vulnerability. `http://lists.gnu.org/archive/html/bug-gnuzilla/2011-06/msg00006.html`.

[6] C. Cadar *el. al.* EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.*, 2008.

[7] H. Chen *et al* . Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS*, 2009.

[8] J. Chow *et al.* Understanding data lifetime via whole system simulation. In *USENIX Security '04*, 2004.

[9] D. D. Clark *et al.* A Comparison of Military and Commercial Security Policies. In *IEEE SSP '87*, 1987.

[10] M. Costa *et al.* Bouncer: securing software by blocking bad input. In *SOSP '07*, 2007.

[11] T. Garfinkel *et al.* Ostia: A delegating architecture for secure system call interposition. In *NDSS '04*, 2004.

[12] I. Goldberg *et al.* A secure environment for untrusted helper applications. In *USENIX Security '96*, 1996.

[13] M. Howard *et al* . Measuring Relative Attack Surfaces. In *WADIS '03*, 2003.

[14] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12th USENIX Security Symposium*, pages 59–74, Aug. 2003.

[15] M. N. Krohn *et al.* Information flow control for standard OS abstractions. In *SOSP '07*, 2007.

[16] N. Li *et al.* Usable Mandatory Integrity Protection For Operating Systems. In *IEEE SSP '07*, 2007.

[17] P. Manadhata *et al* . An Approach to Measuring A System's Attack Surface. Technical Report CMU-CS-07-146, CMU, 2007.

[18] P. K. Manadhata *et al.* An attack surface metric. *IEEE Trans. Software Eng.*, 2011.

[19] J. Newsome *et al* . Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[20] S. Noel *et al* . Efficient minimum-cost network hardening via exploit dependency graphs. In *ACSAC*, 2003.

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

[22] Selinux. `http://www.nsa.gov/selinux`.

[23] X. Ou *et al.* A scalable approach to attack graph generation. In *CCS '06*, New York, NY, USA, 2006.

[24] N. Provos. Improving host security with system call policies. In *USENIX Security '02*, 2002.

[25] N. Provos *et al* . Preventing privilege escalation. In *USENIX Security '03*, 2003.

[26] F. Qin *et al* . LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO 39*, 2006.

[27] S. Rueda, D. King, and T. Jaeger. Verifying Compliance of Trusted Programs. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[28] SecurityFocus. BugTraq Mailing List. `http://www.securityfocus.com/bid/1334`.

[29] U. Shankar, T. Jaeger, and R. Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium*, February 2006.

[30] O. Sheyner *et al.* Automated generation and analysis of attack graphs. In *IEEE SSP '02*, 2002.

[31] W. Sun *et al* . Practical Proactive Integrity Preservation: A Basis for Malware Defense. In *IEEE SSP '08*, 2008.

[32] Sun Microsystems. Trusted solaris operating environment. `http://www.sun.com`.

[33] Tresys. Setools - policy analysis tools for selinux. `http://oss.tresys.com/projects/setools`.

[34] Tresys. SETools - Policy Analysis Tools for SELinux. Available at http://oss.tresys.com/projects/setools. `http://oss.tresys.com/projects/setools`.

[35] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *USENIX ATC '01 FREENIX Track*, 2001.

[36] C. Wright *et al* . Linux security modules: General security support for the Linux kernel. In *USENIX Security '02*, 2002.

[37] N. Zeldovich *et al* . Making information flow explicit in HiStar. In *OSDI '06*, 2006.