The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

RUNTIME MONITORING TOOL FOR UNDERSTANDING ATTACK SURFACES IN PROGRAMS USING SELINUX

A Thesis in

Computer Science and Engineering

by

Guruprasad Gopalakrishna Jakka

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2010

The thesis of Guruprasad Gopalakrishna Jakka was reviewed and approved* by the following:

Trent Jaeger
Associate Professor of Computer Science and Engineering
Thesis Adviser

Bhuvan Urgaonkar
Assistant Professor of Computer Science and Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

ABSTRACT

Attacks on computer systems have become quite common and sophisticated. They exploit vulnerabilities present in programs by injecting malicious code, changing program execution sequence by buffer overflow attacks and essentially affecting the program's integrity. These attacks are launched through the attack surface for the program. An attack surface for a program is the set of entry points which an attacker can use to launch attacks against it. The more features and more complex the program is, the greater its attack surface is. Hence a program with a larger attack surface is vulnerable to more threats from outside. Thus understanding and accurately measuring the attack surface is an important barometer for security of any program. This has been a challenging and an open problem in the security research community.

In this thesis, we build a run time monitoring tool to measure and obtain an accurate assessment of the attack surfaces of the program using its access control policy. Since every entry point for a program could be a potential attack surface, merely identifying it would generate a lot of false positives. We use the integrity wall constructed using the program's access policy to classify MAC policy labels of the program into trusted labels and untrusted labels. This helps in identifying only those entry points which allows untrusted labeled input to enter the program and hence are part of the attack surface. The run time monitoring tool collects all the interfaces where access happens between trusted and untrusted labels. Using this tool, we identify interfaces that have led to several known vulnerabilities. For the `httpd` web server we identified 56 interfaces of which 9 receive untrusted input. These entry points have been associated with a number of vulnerabilities which have been exploited by attackers.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like thank my advisor Prof.Trent Jaeger for his continuous support and guidance, he has shown me during my time here at Penn State.

Chapter 1

Introduction

Programs are subjected to external attacks that try to change the behavior of the program running or corrupt its data. Attacks such as buffer overflow attacks [32], format string attacks, code injection attacks have had crippling effect on present day computing. These attacks form a significant part of all security attacks since they can be easily exploit [27, 24] the vulnerabilities present in the programs.

Many of these programs come bundled with the Operating Systems providing useful functionality to end users. Thus a security attack on a program might affect the entire system on which it is running. Thus the responsibility to protect the system from attacks fall on the OS distributors. To protect the integrity of the system in spite of program compromise, many distributors have started to adopt mandatory access control (MAC) enforcement mechanisms. MAC systems enforces policy on the entire system which cannot be changed by individual subjects. Red Hat and Fedora have started using SELinux [26] as MAC mechanism on their Linux distributions. Microsoft have also introduced a Mandatory Integrity Control [23] in Vista and Windows Server 2008 to restrict permissions to user programs. The MAC policies defined for the programs should ensure the program has access to only those resources which it is required to have. This is in line with the 'Principle of least privilege' where an application is given just enough permissions to have all its functionality. This ensures that when the program is compromised, only those privileges granted to the program are compromised and not the entire system. The policy should defined should make sure that it does not break any functionality for the program it is trying to protect.

Through these MAC policies, the OS distributors are trying to limit their system's attack surface [22]. An attack surface is defined as the set of entry points available to an external attacker to interact with the system. For example a program listening for requests from the network will be part of the attack surface. A program reading files from the file system which can be modified by other subjects will also be part of the attack surface. Some programs are more complex and which might have more interface with external users might be difficult to confine [6]. Thus it is the OS distributors challenge to obtain an accurate understanding of their system's attack surface to protect the system integrity from programs.

The OS distributor would not have enough knowledge about the program to define its MAC policy accurately. Also, the program developer cannot understand the impact of his program on the security of the system. For example, a program reading information from a configuration file might assume the config file to be trusted. He might not add any code for filtering or sanitizing the input read from it. But in the MAC policy for the program, if another process has access directly or indirectly to configuration file then the attacker might use this process to corrupt the configuration file triggering a vulnerability in the program. Thus defining an accurate MAC policy for all the programs protecting system integrity is a challenge.

At present the impact of attack surface has been studied at a system level and individual program level. Analysis tools have been built to understand how well a program is confined by SELinux and AppArmor MAC policies [19, 36, 16, 31, 40, 18]. These tools compute the information flow among programs and the system from the MAC policy defined. This information can be used with different attack scenarios, to study the impact an attacker can cause to the system for the given MAC policy. E.g. if `traceroute` is compromised, it could enable installing a rootkit [6]. Such analysis shows that there are various other attack paths excluding the network facing daemons. These help in identifying the impact on the system of a compromised program but tells us little how to prevent it. Manadhata et al. [21] describe a method to provide a attack surface metric for a program analyzing only the program. They identify all the entry points and

channels for the program to calculate the attack metric. They fail to consider the impact of MAC policy and thus there result will include a lot of trusted interfaces which cannot be exploited.

Using our tool, we develop an approach of using the system's MAC policy for a program to understand its attack surface. We use the integrity wall computed in [37]. This generates a trusted list of subjects and objects. Our tool captures the program instructions responsible for accesses made by the trusted subjects outside integrity wall. We find the exact interface in the program which triggered this access. This interface will be a part of the program's attack surface as it takes in untrusted input with respect to the MAC policy.

1.1 Contribution

The rest of the thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 discusses some of the background concepts and work. Chapter 4 discusses the requirements for the tools and the design we used to achieve it. Chapter 5 describes the implementation details involved for realizing the tool in Linux. Chapter 6 gives the results of the tool running on Apache web server and SSH daemon. Chapter 7 concludes and discusses some of the future work for extending this tool.

Chapter 2

Related Work

In this chapter we summarize work in the area of integrity protection for programs and systems. We also discuss some work such as tainting which can benefit from the work we have done here.

2.1 Taint Tracking

Taint tracking is one approach used to see how data affects the integrity of the program. In these approaches [25, 30, 10, 35, 12, 38, 17, 28] they taint the data which is considered as untrusted and track its propagation through the program. If they are used in a way which affects the integrity of the program, an exception is raised. In all these approaches, they need a way to taint the initial untrusted data. In most of the cases, they taint all the data from network, I/O devices or from the file system. There has not been effort to develop fine grained techniques for tainting untrusted data. The approach we present here can be used to taint the untrusted data in a fine grained manner.

Newsome et al. [25] present a method for automatic detection of attacks and generating filters automatically using dynamic taint analysis. In this approach they taint all the data obtained from untrusted sources and data derived from these untrusted sources. They then track this data and see if they are used in dangerous ways such as changing the function pointer, return address and other control flow data. To perform this they need to keep a shadow memory which tracks the taint for every byte of memory in the memory. The performance overhead for such approaches is around 20 times more than native execution for average cases.

Garfinkel et al. [8] also use tainting to analyze the life times of sensitive information in programs. They taint all the sensitive data and propose taint propagation rules to tag data derived from it. They perform whole system logging to track where the sensitive data originated, how long the data is present and its progression through the program. To identify the source of taint data they specify a coarse method of tagging all input from keyboard or network devices. They also introduce a new instruction to IA-32 architecture to specify that a particular data is tainted. This is difficult to realize in practice. Our method can be modified and used to tag all the trusted data which taintbochs can use further.

Suh et al. [35] uses hardware mechanism to tag data from spurious channels and tracks the flow of this data through the program. Any data which is related or copied from spurious data is not allowed as an instruction or jump target instruction. Identifying spurious channels is done by the OS at an I/O Level by tagging all network I/O or file I/O. No distinction is made among files which are trusted by the program and files which are untrusted. In our tool, we use SELinux Labels to determine the trustworthiness of a file. We could employ some of the techniques used to track the flow of spurious data to our tool after we identify potential interfaces for entry of bad data.

2.2 Control and Data Flow Integrity Approaches

Enforcement of control flow integrity and data flow integrity have been proposed to maintain integrity of programs. Control flow integrity approaches try to control attackers from arbitrarily changing program behavior to follow a path a Control Flow Graph (CFG) determined ahead of time. CFGs can be generated using source code analysis, binary analysis or execution profiling. Attacks such as Buffer overflow occurs when the return address is changed by the attacker. Since this changes the control flow of the program it would be detected by mechanisms enforcing CFI [14, 13, 3, 29]. Abadi et al. [2] ensures program security by ensuring that only

legal flow by a program is allowed during any run of the program. However non-control-data attacks [7] may not alter the control flow and hence may not be detected.

Data flow integrity approaches ensure integrity of data by making sure that data read was modified only by valid instructions. Castro et al. [5] builds a static data flow graph which ensures that any data read was modified by code from valid locations in the program. This helps in avoiding control-data attacks and non-control data attacks. Yip et al. [39] identifies that the code for handling and sanitizing data exists at various places in the program. These checks are necessary for integrity verification of the data and proper authorization to access the data. It avoids this approach by associating security checks to particular chunks of data. Any attempt to access the data invokes the security checks. These approaches detect at attack too late and it requires the program to restart. They detect the attack but the program will not know how to handle it without restarting. Our approach is mainly to help program developers handle untrusted data in the first place by understanding the threat model with respect to its MAC policy.

2.3 Integrity Protection Models

Integrity models such as Biba [4], Clark-Wilson [9], LOMAC [15], Lipner specify rules which subjects should follow when interacting with objects. In Biba, a set of totally ordered integrity levels are defined by the system. Every subject and object in the system is assigned an integrity level. A process belonging to a integrity level can read from objects of equal or higher integrity level. It also enforces that a process cannot write to an object of higher integrity level than itself. This ensures that no information flows from a low integrity objects to high integrity objects. This model is difficult to emulate in practice since processes invariably have to read data from low integrity objects. In Clark-Wilson a process can read from low integrity objects but the interfaces which read such data have to be certified so that it discards or upgrades the untrusted data suitably. Shankar et al. [34] develop a model, CW-Lite which satisfies information flow

integrity guarantees by processes and develop automated tools for practical verification. This requires modifications to the programs by adding annotations to filtering interfaces manually by the developer. Our technique can be used for identifying the filtering interfaces automatically.

The integrity rules defined by these models have to be enforced by the OS and cannot be modified by processes themselves. Thus, Mandatory access control mechanisms are used to enforce these integrity models. Li et al. [20] design a Mandatory Integrity Protection protecting the system integrity against network based attacks. It adds a usable mandatory access control system to the OS which can be deployed easily. They protect the hosts when privileged network facing programs containing vulnerabilities are exposed by attackers. SELinux [26] implements an mandatory access control system for the Linux kernel. In a well configured system, it would help protect the system integrity in the case of a program getting compromised. However, it may not help protect the program's integrity. There may be rules specified in the policy which might allow untrusted input to be read by the trusted subjects in the program.

Jaeger et al. [19] identify a minimal TCB for the program and check the SELinux policy for any conflicts with the system's security goals. This is a static analysis of the policy for verifying its integrity protection. We can adopt this technique of identifying the TCB for the program in our tool to log the violations.

Manadhata et al. [21] introduces attack surface metric to measure the attack surface using I/O automata model of the system. They identify the list of library calls allowing data to enter the system to calculate their metric. This does not take into consideration whether inputs are trusted or untrusted. In our analysis, we found that a significant number of interfaces receive high integrity input and hence the vulnerabilities in these interfaces cannot be exploited by external attackers. Also, it only gives a attack metric and does not help in identifying locations in the programs which might be vulnerable. Our tool helps in identifying the locations in the program excluding any library calls where an exploit can occur.

Bouncer et al. [11] is a system which uses the knowledge of vulnerabilities to detect attacks and generate filters automatically. It prevents the attack by dropping exploit messages before they are handled by the program. Bouncer could use our knowledge of vulnerable interfaces of the program, and generate filters for inputs that cause failure.

Chapter 3

Background

In this chapter, we briefly discuss the approaches taken for securing systems focusing on integrity protection. Confidentiality, Integrity and Availability are the three main components of computer security. Confidentiality deals with ensuring that a resource or data is not leaked or made available to a subject which is not supposed to. It deals with restricting access to subjects by prohibiting unauthorized disclosure of information. Confidentiality policies ensure that data from a higher secrecy level is not leaked to a subject of lower secrecy level. Availability ensures that a given resource is available for use when desired. Integrity refers to the trustworthiness of the resource in consideration. The integrity of the resource is defined initially by the source of its origin. If the subject which creates a resource is of high integrity, then we give the resource high integrity. Resources which are created by unknown sources with respect to the system will be labeled as low integrity. When a resource of higher integrity is modified by a subject of lower integrity, we say that the resource is corrupted. Any assumptions made by the system assuming the resource is of higher integrity will be invalidated. Integrity policies ensure that a integrity of a resource is not compromised during its life time.

3.1 Information Flow Security Models

Information flow can be used to specify the secrecy and integrity requirements of a system. Information flows occurs between a subject and an object when a subject reads or writes to an object. There is a flow from the object to subject when the subject reads from the object and there is a flow from subject to object when the subject writes to the object. Transitively an information flow occurs between two objects when a subject reads from one object and writes

to another object. Information flow policies classify information into various categories or levels and specify what flows among objects are valid and what flows are invalid.

3.1.1 Bell-La Padula Model

The information flow model for secrecy based on Multi Level Security was specified by Bell and La Padula (BLP). In this model, subjects and objects are assigned a security level. These security levels follow a total order. For example consider a system with 4 secrecy levels Unclassified (UC), Confidential (CO), Secret (S) and Top Secret (TS). The resource with a secrecy level TS dominates the resource with a secrecy level S. The BLP policy specifies the valid information flows across subjects and objects in the system satisfying the two properties

- Simple Security Property a subject at a given integrity level cannot read objects at higher secrecy level
- * (star) Property a subject at a given integrity level cannot write to an object of lower secrecy level.

These are referred to as no read up and no write down properties. If the policy is properly enforced, it would prevent leak of information from a higher secrecy level to a subject of lower secrecy level.

3.1.2 Biba Integrity Model

Biba [4] specifies an information flow model for integrity which is a dual of Bell-LaPadula model. As in the BLP model, all the subjects and objects are given integrity labels. These integrity labels are totally ordered. It specifies two properties

- Simple Integrity Property a subject at a given integrity level cannot read objects at a lower integrity level.

- * (star) Integrity Property a subject at a given integrity level cannot write to objects at a higher integrity levels.

There are referred to as no read down or no write up properties. If the Biba model is specified and implemented properly on a system, it would prevent a resource or object of high integrity ever getting corrupted. This is model is not feasible to implement in most programs since high integrity process invariably needs data from untrusted sources to function. For example, a web server is a high integrity process which needs to accept data from untrusted sources from outside.

3.1.3 Clark -Wilson Integrity Model

Clark and Wilson [9] specify an integrity model which is more clearly applicable to business and industry processes in which the integrity of the information content is more important than maintaining confidentiality. It defines objects in the system whose integrity needs to be enforced as Constrained Data Items (CDI) and objects whose integrity need not be enforced as Unconstrained Data Items (UDI). A set of integrity constraints is defined which defines the legal values for the CDIs. The model also defines two set of procedures. Integrity Verification Procedures (IVP) checks if the CDIs conform to the integrity constraints when the IVPs are run. Transaction Procedures (TP) take the system from one state to another state ensuring that the CDIs still conform to the integrity constraints defined. In this model, high integrity subjects can read input from low integrity objects if they are certified to upgrade the input data or discard the data. CW allows the programs to handle low integrity data as long as they are certified that they would handle the incoming data securely and not affect the integrity of other data objects. In practice, certifying programs in an automated manner to check if all the program entry points satisfy the requirements to handle data securely have not been developed. CW lite [34] is another model proposed for ensuring information flow integrity. It is a weaker model than CW but ensures the same interprocess dependency semantics. CW-Lite requires filtering for untrusted interfaces depending on the system security policy.

3.2 Security Vulnerability

We say that a program has a vulnerability [1] if an attacker can activate a flaw in the program and change its expected behavior. The most common type of vulnerabilities are the remote network penetration vulnerabilities. This allows an anonymous user in the Internet to obtain partial or complete control of the host system. A vulnerability that has one or more working attacks is called an exploit. Some of the attacks which have exploited vulnerabilities in systems include buffer overflow attacks, format string attacks, cross site scripting, SQL injection and many others. These attacks can leak sensitive information compromising confidentiality of the system and also corrupt data in the system affecting its integrity and availability. Various approaches have been developed to prevent exploitation of vulnerabilities in existing code. Some of these require changes to the source code by instrumenting checks and filters before sensitive system calls and others work on the binaries of the program. Here we briefly discuss some important attacks.

3.2.1 Buffer Overflow Attack

Buffer overflow attacks have been the most common exploit in recent years. An buffer overflow happens when data being written into a buffer exceeds the capacity of the buffer and overwrites into the memory adjacent to the buffer. This happens if there are no bound checks while performing copy. When the overwritten memory is a control information like return address on a stack, the control of the program is altered from its intended behavior. Carefully crafted overflow involves overwriting the control information to execute a malicious code introduced by the attacker or perform an operation which is not authorized. Exploits on stack and on heap are the most popular ones. Protection schemes for stack involve non executable stack and stack canaries. Stack canaries involve placing a random integer before the return address pointer. If the buffer overflows into return address pointer, the canary would also be overwritten.

The canary is checked before using the return address to return from the stack. Other solutions include randomizing address space [33], executable space protection, use of safe libraries, pointer protection [13].

Chapter 4

Design

This chapter describes the requirements and main structure of the run time monitoring tool. Our tool is used to measure attack surfaces for programs and systems. This requires setting up the system with the MAC policy for the program and also developing a policy for the tool to decide when to capture access. We use the integrity wall construction described in [37] as input for our tool.

4.1 Integrity Wall Construction

For measuring the attack surface of programs, we use he MAC policy defined for the program by the system administrator. Using the MAC policy provides a much finer understanding of the attack surface and eliminates a lot of entry points which cannot be exploited by attackers without the compromise of the system. We use the method described by Vijayakumar et al. to build an integrity wall for the application subject labels. A Trusted Computing Base (TCB) as to been defined for building the integrity walls. The TCB for the program would include subjects labels from the system and subject labels from the application itself. There is no standard method for defining the TCB except that it should satisfy properties like having a smaller size, the need to protect itself and the requirement that if the TCB is protected, then the rest of the system can be protected.

For the system TCB, they include a small number of subject labels which are necessary for initialization of the system, MAC policy administration, providing core kernel functionality. For constructing the application TCB, they include those subject labels whose executables are only modified by the system TCB subject list. This would ensure that all the code which runs for

the application comes during installation time or updated by high integrity subjects. It would exclude any subject label whose executable could be modified by lower integrity subjects. Once, the trusted subject lists from the system and the MAC policy are computed, the trusted object labels list is derived. It includes all the object labels which have all their write like accesses by the trusted subject lists. Using this criteria, the objects are classified into trusted and untrusted lists. Any read flow from the untrusted list to trusted list would constitute an access across the Integrity wall.

4.2 Identifying Untrusted Operations

Once we obtain the integrity wall for a subject label, we want to determine if a process for the program accesses objects which are of lower integrity or outside its wall, called untrusted operations. This can be identified by statically and dynamically. The MAC policy itself might allow flows from lower integrity labels to higher integrity labels. It might be necessary for the program to read from network sockets receiving data from outside. The program might also read from configuration files which can be modified by users of the system. In these cases, the MAC policy itself is allowing flows across the integrity walls. But it does not inform anything about the location where the flow is occurring.

With dynamic approach we can identify the exact location in the program which would cause the untrusted operation. We would like to monitor for every untrusted operation during the program execution. The criteria for effective monitoring are below.

- It should act as a reference monitor mediating all security critical operations by all processes in the system
- The subject and object labels for each of the access by the subjects
- The context of the program execution, the instruction responsible for the access.

Effectively, it should act as a reference monitor. It should be able to mediate every security critical operation by any process in the system on any object of any type. The tool itself cannot be bypassed by any process directly or indirectly. It should be small enough for manual verification. The trusted subject list and the accesses logged by the tool should also be protected from tampering.

To perform the monitoring of all the accesses, we would need the subject and object labels involved in the operation. Using the integrity wall constructed earlier, the tool can make a decision observing the labels in the operation if the access needs to logged or not.

The context of the process execution, line in the code generating this access also need to be available. This would be the entry point in the program where a probable attack can be launched.

4.3 Finding Entry Points

Figure 4.1 shows the steps involved in finding the entry points. Once the untrusted operation is found, we want to determine the program instruction responsible for this operation. These would be the entry points for untrusted data from the attackers for the program with the given MAC policy. Finding the exact program instruction is a bit involved. The monitoring for accesses is done in kernel, but the instruction would be in the processes' address space. Since kernel has access to all the processes' address space, we can access the address space of the process responsible for the flow.

In most of the programs, the system call for performing an operation is rarely called directly by the user program. It is interfaced by a library or set of libraries which perform additional book keeping and providing more functionality than the system call itself. For example, Apache web server uses `libapr` library which in turn may call `libc`. Thus when we locate the program instruction responsible for the untrusted operation, it will invariably point to the system call in the `libc` library responsible for the access. This does not provide any context with respect

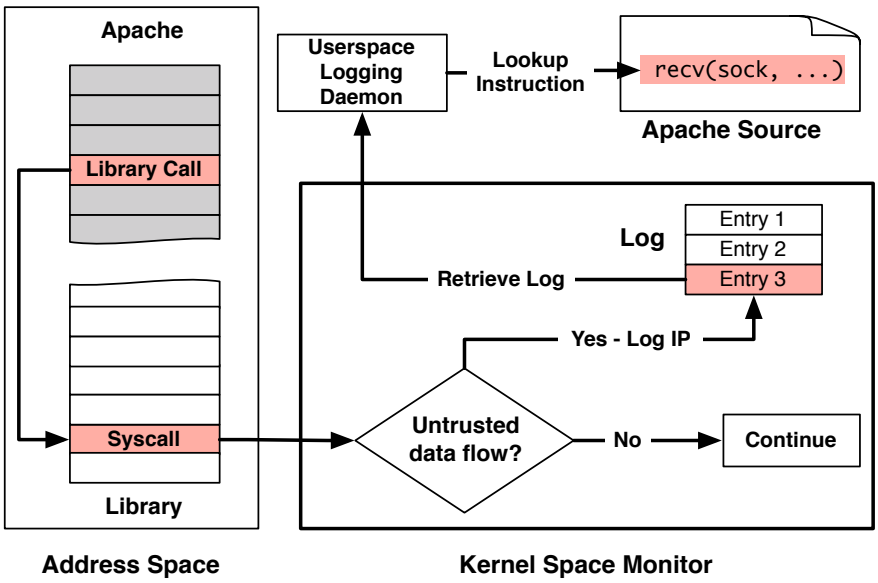

Fig. 4.1. In the Apache address space, an instruction may perform, through a series of library calls, an untrusted operation (i.e, read data from a network socket). The kernel space monitor detects whether the operation is untrusted and logs the instruction pointer, which is later sent to a userspace daemon. The pointer is then correlated with debug information to find the instruction in the source code.

to the program's entry point. We need the library call in the program which triggers the system call. For this, we need to look into entire call stack looking for the last instruction called from the program. The last few frames would belong to the invocation of code inside libraries. We need to remove all the library contexts from the call stack trace and retrieve the last frame from the stack for the user program. To do this, we must find a way to distinguish code segment addresses belonging to libraries and the program executable.

4.4 Logging and Summary Statistics

Once we identify the untrusted operation and extract the program context we need to store these entries. Since we perform this operation in kernel space, it needs to be transferred to user space. We build the summary statistics containing interfaces in the program and the number of times a flow occurred from low to high integrity. We also need to separate overt flows (E.g. `FILE__READ`) and covert flows (E.g. `DIR__SEARCH`). The interface with high count number indicates the usage of this path for flow during the particular run of the program. To obtain most of the interfaces for allowing bad information flow, we should run the user program in all of the possible scenarios so that it explores most of its code path. If we would like to analyze all the interfaces in the program, we can remove the check in the kernel to log access to untrusted objects. The tool would then record all accesses by the trusted subjects.

Chapter 5

Implementation

In this chapter, we discuss the implementation of our run time monitoring tool. We implemented the design on a 2.6.30 Linux Kernel with our modifications running Ubuntu 8.04 Server Edition on an Dell Optiplex 745. SELinux was used as a MAC enforcement mechanism.

5.1 Enforcement Point

The main requirement of our tool is that it should be able to mediate all security critical operations from the untrusted to trusted labels. Since we are using SELinux as the MAC system, we place it in the location where SELinux enforces its MAC policy for the system. '`avc_has_perm`' is the main function for access check for SELinux which authorizes if a subject label can perform an operation on the object label. We instrument this function to enforce our check for accesses across integrity wall. The trusted subject lists is exported from a debugfs file into the kernel. It is a list of system TCB types. In order to measure attack surfaces for multiple programs during a single run, we avoid loading the non system trusted object types for the programs into kernel. This is because, trusted object types for one program might be untrusted for another program. We record all read like operations from the these trusted subject lists. We use user space tools to filter out accesses by different programs and accesses to non system trusted object types.

5.2 Obtaining Process Context

Figure 5.1 shows our overall implementation. Once the flow from untrusted label to trusted label is caught, we need to capture the program instruction responsible for this flow. This indicates the access across the integrity wall. We would need the process context of the instruction

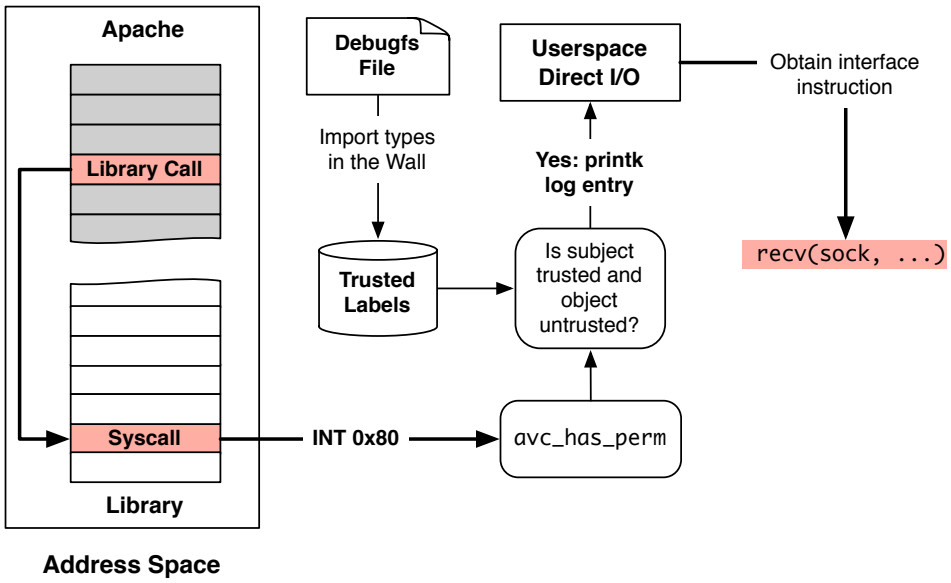

Fig. 5.1. When the program makes a syscall via the INT 0x80 interrupt, the SELinux function `avc_has_perm` is hooked. If the calling function causes an object with a label not in the trusted label set to be read by a trusted process, then the stack trace is printed to userspace. The entry is later processed to obtain the instruction of the interface. The trusted label set is pushed into the kernel via a debugfs file.

responsible for the access. Since we are inside the kernel, we have sufficient permissions to access the context of the current running process.

The user stack for the process contains the back trace of the running process. It is not accessible from the current stack pointer (`esp`) as it points to the stack in kernel space. When the process enters the kernel to execute the system call responsible for the flow, the user stack along with the register context for the process are captured and stored. In Linux, it is stored in a 4KB page adjacent to '`task_struct`' structure for that process. We use the '`current`' macro which points to the '`task_struct`' of the current running process to get access to the user stack pointer stored in the kernel stack. Recording the stack trace is complicated in new processors (Intel Pentium III onwards). Traditionally, system calls have been invoked by a trap instruction (INT 0x80) from the user space. This was observed to be an order of magnitude slower on the newer processors compared to old x86 processors. So, a new and faster way to invoke system calls was supported using the `SYSENTER` instruction. This was implemented in Linux using the Virtual Dynamically Linked Shared Object (VDSO) library. However, we found this obfuscates the instruction pointer for the last frame in the program. Hence, we disabled the VDSO option for our kernel.

We need the whole function call trace to obtain a accurate understanding of the process context. In the x86 architecture, the 'ebp' register is the base frame pointer which is used to access information of the last frame in the stack trace for the process. Using the 'ebp' pointer we can extract the current instruction being executed by the process and go up the stack for instruction pointers for all the frames. This gives the process context responsible for generating the flow from untrusted to trusted label. From 2.6.28 kernel onwards, function tracer - ftrace a tracing infrastructure was introduced inside the Linux kernel. It is used to debugging and analyzing performance issues inside the kernel. We use the ftrace code to obtain the stack trace for the process running. It also provides an option to specify how much depth of the user stack is required.

5.3 Removing the Library contexts from the Stack Trace

We need to remove the instruction pointers belonging to the libraries and return the last instruction pointer in the frame for the user program. In Linux every memory address is mapped to a region in the process address space. These address spaces belong to various object files including the executable of the user program, the library executable, kernel stack, heap. We use the virtual memory address structures in Linux to identify the first instruction pointer from the call stack which maps to the executable of the user program.

5.4 Logging the flow violations

Once we detect the flow from untrusted label to trusted label and capture the process context of the access, we need a way to store this information, so that it can be retrieved at a later point of time. Since, we are obtaining the context information inside the kernel, we need to transfer this information to user space for further analysis. Also, this logging information is produced at a very high rate which the logging mechanism should be able to handle. The `ftrace` mechanism had an infrastructure to transfer its tracing information back to user space, but we were unable to find an proper API to plug into. We use the `printk` logging mechanism to facilitate this transfer. Due to the amount of information generated, the `printk` logging system was overwhelmed and some of our data was getting garbled. So we built our own rate limiter inside the kernel using `procfs`. For every access that needs to be logged, we create a structure which holds this information and stores it in a linear linked list inside the kernel. We don't transfer this information directly to the user-space. We create a file in the `procfs` system, which is used as a trigger to start transferring log entries from kernel to user space through printk. For every read of the `procfs` file, we go through the linear linked list and transfer a certain number of entries (avoiding overwhelming `printk`) into the user space using `printk`. We tag the information we transfer using the `printk` function.

5.5 Analyzing the Interface Information

We extract the entries from the `printk` buffer to a user space file using the 'sed' command. This contains an entry for every single read like access performed by our trusted subject types. A typical entry 5.2 consists of the tag, serial number, process id, process name, flag indicating if the address in the user program was found, call trace eliminating the library frames, subject label, object label for the access, SELinux class number and the operation requested.

```
 TR,3905),13553,sshd,(program_ip: 1),807d150,8048000,
(unconfined_u:system_r:sshd_t:s0-s0:c0.c1023,system_u:system_r:getty_t:s0,8,1)
```

Fig. 5.2. A sample entry captured for an untrusted operation in the kernel

The entry provides us with all the information for further analysis. We have one more problem in that we need to identify whether the operation is read like or write like. For this, we use the permissions file from `apol` file to filter the read like operations. This file also provides with a number for each operation. Larger number indicating more overt flow. This is helpful in ignoring a lot of small covert flows like `DIR_SEARCH`, which does not help us in understanding the attack surface for the system. We group all accesses belonging to a particular process together. Then we group based on the last code segment address in the stack. This is the entry point for the user program where the untrusted flow happens. We list out all the accesses which occur with the same access type, subject and object type.

The figure 5.5 shows a sample output generated by our user space tool. It lists the file name and line number associated with the Instruction Pointer. We use the '`addr2line`' command

with the binary compiled with source information of the program to generate the file information. We group them by different object types and by the operation type of the access.

```
PROCESS: httpd  CONTEXT: unconfined_u:system_r:httpd_t:s0
Number of filtering interfaces: 51
{
   807cdf0, /usr/local/src/httpd-2.2.3/server/config.c:1960,   1960
       1, NETLINK_ROUTE_SOCKET__GETATTR, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066bcc', '8065e11']
       1, NETLINK_ROUTE_SOCKET__NLMSG_READ, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066bcc', '8065e11']
       3, NETLINK_ROUTE_SOCKET__READ, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066bcc', '8065e11']
       1, NETLINK_ROUTE_SOCKET__GETATTR, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066d59', '8065e11']
       1, NETLINK_ROUTE_SOCKET__NLMSG_READ, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066d59', '8065e11']
       3, NETLINK_ROUTE_SOCKET__READ, unconfined_u:system_r:httpd_t:s0,
['807cdf0', '8066d59', '8065e11']
}
```

Fig. 5.3. Sample output of our tool listing the interface information which receives input from untrusted object types

Chapter 6

Evaluation

We discuss the results from analysis of running our tool on Ubuntu 8.04 Server Edition with the SELinux Policy running 2.6.30 Linux Kernel. First we discuss the performance impact the tool has on general system calls. The processing involved includes two steps. First one is to check if the subject and object for the access needs to be logged. This involves checking if the operation results in an access outside the integrity wall. The second step is to actually capture the user stack trace of the process, remove the library function calls and log the process. We have calculated the time taken for these each of these two steps.

From 2.6 kernel onwards, the kernel code is premptible. Hence, the kernel thread might get context switched out while executing our code. So, we set the config option for the kernel `CONFIG_PREEMPT = NO` and built our modified kernel. This option makes the kernel non premptible ensuring that the filtering code run uninterrupted. We collected the times for several hundred system calls and averaged it.

Operation	Time Taken (msecs)
Access Check	111
Stack Capture and Logging	118

Table 6.1. Time taken by the tool for Capturing accesses

As seen from the table, we find the time to perform the identification and capture of accesses outside of integrity wall is quite small and its impact on the overall running time of the system is quite minimal.

Next, we performed analysis on two programs - `httpd` and `sshd`. We used the SELinux policy and the trusted types from [37] for our experiments. The trusted types had two components the system TCB types and the trusted types for the application. System TCB types included all the subject types which are considered part of the Trusted Computing Base for the system. The SELinux policy had a total of 1085 (subject and object) types. The system TCB included 111 subject types.

6.1 Runtime Analysis for Apache Web Server

Untrusted type	Description
httpd_user_script_exec_t	User-defined scripts, modifiable by user_t
httpd_user_script_t	User-defined scripts
httpd_user_content_t	Webpages created by User
httpd_user_htaccess_t	User-defined configuration files
httpd_log_t	HTTP log file which could be written to by httpd_user_script_t

Table 6.2. Apache untrusted object types.

We ran the Apache web server (ver 2.2.14) with mod_perl on our modified kernel to evaluate its attack surface. We used `Apache::Test` perl module, which contains scripts for testing the core functionality of the `httpd` web server. This is used by Apache developers before release of a new version to the field. There are many other testing suites for `httpd`, but most of

them focus on performance and load testing. While these might be helpful, they might miss out of testing many of the rarely used functionality and options.

After executing the test suite on the webserver, our tool recorded **14868** accesses. These included both read like and write like accesses. It also included accesses made by trusted subject types to trusted object types. This number gives an idea about the program's interaction with the system and the environment during a typical usage. The total read like accesses was **14520**.

ID	SLOC	Associated Types	Description
1	server/util.c:904	httpd_user_htaccess_t	open user .htaccess file
2	server/util.c:879	httpd_user_htaccess_t	read user .htaccess file
3	server/listen.c:140	httpd_t	socket listen
4	os/unix/unixd.c:506	httpd_t	tcp socket accept
5	server/core_filters.c:155	httpd_t	read from socket
6	server/core.c:3694	httpd_user_content_t	open user HTML file
7	server/core_filters.c:383	httpd_user_content_t	read user HTML file
8	server/connection.c:153	httpd_t	read remaining data from socket
9	os/unix/unixd.c:410	httpd_user_script_exec_t	execute CGI user script

Table 6.3. Apache interfaces that receive low-integrity data

The untrusted object types obtained after integrity wall construction are listed the table 6.2. The number of program interfaces responsible for the accesses were **56**. This represents the 'attackability' of the program. Of these only **9** interfaces had flows from untrusted to trusted types. List of untrusted interfaces is presented in the Table 6.3. Rest of the interfaces only accept data from trusted objects and hence can be considered to be safe. This shows for a typical run of the program, what interfaces are more likely to contain vulnerabilities. Identifying all the library interfaces of a program would result in a significant overestimation.

6.1.1 Analysis of Interfaces Receiving Untrusted Data

- Interfaces 1 and 2 in table 6.3 identifies the location where the .htaccess file is read by the program. It is the configuration file which can be modified by users to set per directory custom settings. Since the user has complete control over the content of the file, the data read from here cannot be trusted. There have been several vulnerabilities due to incorrect parsing of the .htaccess file. (CVE 2009 1195, CVE-2005-2491, CVE-2004-0747, CVE-2003-0542).

- Interfaces 3, 4 and 5 in table 6.3 are main locations in the program which takes in requests from outside users to service their web requests. An maliciously crafted HTTP requests can exploit the vulnerabilities in parsing requests by the web server.

- Interfaces 6 and 7 in table 6.3 identifies the location where user created web pages are read by the program. A malicious web page could exploit a vulnerability if filters are not put in place while parsing the web pages.

6.1.2 Observation of the Interfaces captured

```
812aa38, httpd-2.2.14/os/unix/unixd.c:420
          5, FILE, FILE__EXECUTE, user_u:object_r:bin_t:s0,
          5, FILE, FILE__EXECUTE_NO_TRANS, user_u:object_r:bin_t:s0,
          15, FILE, FILE__READ, user_u:object_r:bin_t:s0,
          5, LNK_FILE, LNK_FILE__READ, system_u:object_r:lib_t:s0,
          5, FILE, FILE__EXECUTE, system_u:object_r:ld_so_t:s0,
          10, FILE, FILE__READ, system_u:object_r:ld_so_t:s0,
```

Fig. 6.1. An example interface which reads data from multiple object types and performs multiple operations

We also observed that for a single interface, there were more than 1 kind of access - `FILE__EXECUTE`, `FILE__READ` and `LNK_FILE__READ`. A single interface also had accesses to different object types including trusted and untrusted types. Presence of such interfaces can alert the developer of the program to be extra cautious while writing any filtering code for sanitizing input data to ensure that data from all the types are handled securely.

6.2 Analysis on Secure Shell Daemon

We also performed a study on SSH daemon.Our tool logged 3257 accesses of which 2965 were read-like accesses. There were 77 interfaces which was caught of which 27 accepted untrusted inputs. The table 6.2 lists these interfaces which allow overt flows.

ID	SLOC	Associated Types	Description
1	monitor_wrap.c:133	sshd_t	Unix socket read
2	sshd.c:1719	sshd_t	TCP socket listen
3	sshd.c:1749	sshd_t	TCP socket accept
4	msg.c:72	sshd_t	Unix socket read
5	msg.c:84	sshd_t	Unix socket read
6	sshd.c:442	sshd_t	TCP socket read
7	monitor_wrap.c:123	sshd_t	Unix socket read - master interface
8	dispatch.c:92	sshd_t	TCP socket read
9	packet.c:1005	sshd_t	TCP socket read
10	auth.c:519	user_home_ssh_t	user .ssh/ file open
11	sshpty.c:63	ptmx_t	open pseudo terminal for user
12	channels.c:1730	ptmx_t	read from pseudo terminal
13	serverloop.c:380	sshd_t	fifo file read
14	loginrec.c:1423	initrc_var_run_t	read utmp

Table 6.4. SSHd interfaces that may receive low-integrity data

Chapter 7

Conclusion

In this work, we developed a tool for identifying and measuring the attack surface of programs and systems with respect to its MAC policy. It avoids the oversimplification approach by many who measure all the entry points for the programs as its attack surface. It can used by program developers to have a better understanding of the threat model for the program with respect to the system it will run. We implemented the tool satisfying all the reference monitor guarantees. Though we have implemented the tool in SELinux on Linux kernel, the design could be used to implement the same behavior in other operating systems and on any MAC enforcement mechanism. We were able to find out the total number of interfaces a program is exposing and the vulnerable interfaces which actually read from untrusted labeled input. We also classified the interfaces which allow more overt flows than simple covert flows. This gives a better understanding of the security of the program for the administrator deploying the program. Also, it helps the developers of programs to go back and verify the interfaces which accept untrusted input. They can then ensure that the input is properly sanitized and filtered before using it further. Few of the interfaces found were linked to several known and recently discovered vulnerabilities. The trusted subject list can be modified suitably to suit the administrator or developer's sense of the security and find out the attack surfaces accordingly.

As a part of future work, the tool can be extended to maintain separate subject and object label list for each program and use this to log only pertinent entries. It can be also modified to incorporate write like operations, focusing on secrecy of the data in an application. It can also be integrated with other techniques such as taint tracking or slicing to track the propagation of the

data received from the untrusted labels and identify secondary interfaces in the program where this data can be exploited.

Appendix

Trace Capture and Analysis

A.1 Capturing the Trace

Insert a call to the function - `record_trace()`, where the trace has to be captured. This captures the back trace of the user program whenever the `record_trace()` function call is executed.

A.2 Obtaining Trace to a Log File

A directory entry is created in the `procfs` system - `'selinux_violations'`. The directory has a file name 'list'. When we read this file, a certain number (currently 100) of trace entries gets printed into the `printk` log buffer. We can repeatedly read this file and transfer all the traces from the kernel to the `printk` log buffer. Next, we need to transfer the `printk` log to a trace file. We use `'dmesg'` to perform the operation.

```
cat /proc/selinux_violations/list

dmesg -c >> trace.txt
```

The trace.txt file would contain the entire `printk` log. We need to filter our trace messages from it. The trace entries are tagged with "TR" string. We use `'sed'` program to extract the trace entries and place it into a separate file.

```
sed -e s/\<7\>//g trace.txt | grep -a "TR," > trace_real.txt
```

A.3 Format of trace entries

```
[  141.553991] TR,4),5704,httpd,4,807cdf0,8066bcc,b7ce2685,8065e11,
(unconfined_u:system_r:httpd_t:s0,unconfined_u:system_r:httpd_t:s0,43,16)
```

Tag for the trace entries - "TR"

Counter - Number of Trace entries - "4"

Process ID - "5704"

Process Name - "httpd"

Number of stack trace entries - "4"

Stack Trace entries - "807cdf0,8066bcc,b7ce2685,8065e11"

Subject Type of the Process - "unconfined_u:system_r:httpd_t:s0"

Object Type requested - "httpd_t:s0,unconfined_u:system_r:httpd_t:s0,43,16"

References

[1] Vulnerability (computing). `http://en.wikipedia.org/wiki/Vulnerability_(computing)`.

[2] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security'05*, pages 340–353, 2005.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *12th USENIX Security Symposium*, Washington, DC, August 2003.

[4] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, MITRE, April 1977.

[5] Miguel Castro. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 147–160, 2006.

[6] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and comparing the protection quality of security enhanced operating systems. In *NDSS*, 2009.

[7] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium*, pages 177–192, 2005.

[8] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[9] D. D. Clark and D. Wilson. A Comparison of Military and Commercial Security Policies. In *1987 IEEE Symposium on Security and Privacy*, May 1987.

[10] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[11] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 117–130, New York, NY, USA, 2007. ACM.

[12] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain internet worms?, November 2004.

[13] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard trade: Protecting pointers from buffer overflow vulnerabilities. In *In Proc. of the 12th Usenix Security Symposium*, 2003.

[14] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Symposium*, pages 63–78, 1998.

[15] Timothy Fraser. Lomac: Low water-mark integrity protection for cots environments. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

[16] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying Information Flow Goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[17] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 29–41, New York, NY, USA, 2006. ACM.

[18] Hong Chen and Ninghui Li and Ziqing Mao. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *NDSS*, 2009.

[19] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[20] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 164–178, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Pratyusa Manadhata, Kymie Tan, Roy Maxion, and Jeannette M. Wing. An Approach to Measuring A System's Attack Surface. Technical Report CMU-CS-07-146, School of Computer Science, Carnegie Mellon University, 2007.

[22] Michael Howard and Jon Pincus and Jeannette M. Wing. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security*, 2003.

[23] MSDN. Mandatory Integrity Control (Windows). `http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx`.

[24] "Mudge". How to write buffer overflows. 1997.

[25] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *In Proceedings of the Network and Distributed System Security Symposium*. Internet Society, 2005.

[26] Security-Enhanced Linux. `http://www.nsa.gov/selinux`.

[27] "Aleph One". Smashing the stack for fun and profit. 7(49), November 1996.

[28] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[29] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, pages 211–224, 2003.

[30] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148. IEEE Computer Society, 2006.

[31] B. Sarna-Starosta and S.D. Stoller. Policy Analysis for Security-Enhanced Linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004.

[32] Fred B. Schneider, Steven M. Bellovin, Martha Branstad, J. Randall Catoe, Stephen D. Crocker, Charlie Kaufman, Stephen T. Kent, John C. Knight, Steven McGeady, Ruth R. Nelson, Allan M. Schiffman, George A. Spix, and Doug Tygar. Trust in cyberspace. 1999.

[33] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[34] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the Network and Distributed System Security Symposium, (NDSS 2006)*, 2006.

[35] G.Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96. ACM, 2004.

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

[37] Hayawardh Vijayakumar, Guruprasad Jakka, Sandra Rueda, Joshua Schiffman, and Trent Jaeger. Integrity Walls: Finding attack surfaces from mandatory access control policies. Technical Report Technical Report NAS-TR-0124-2010, Network and Security Research Center, February 2010.

[38] Jun Xu and Nithin Nakka. Defeating memory corruption attacks via pointer taintedness detection. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, pages 291–304, 2009.

[40] Giorgio Zanin and Luigi Vincenzo Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145. ACM, 2004.