Policy Models to Protect Resource Retrieval

Hayawardh Vijayakumar, Xinyang Ge, and Trent Jaeger
Systems and Internet Infrastructure Security Laboratory,
Department of Computer Science and Engineering,
The Pennsylvania State University
{hvijay,xxg113,tjaeger}@cse.psu.edu

ABSTRACT

Processes need a variety of resources from their operating environment in order to run properly, but adversary may control the inputs to resource retrieval or the end resource itself, leading to a variety of vulnerabilities. Conventional access control methods are not suitable to prevent such vulnerabilities because they use one set of permissions for all system call invocations. In this paper, we define a novel policy model for describing when resource retrievals are unsafe, so they can be blocked. This model highlights two contributions: (1) the explicit definition of adversary models as *adversarial roles*, which list the permissions that dictate whether one subject is an adversary of another, and (2) the application of data-flow to determine the adversary control of the names used to retrieve resources. An evaluation using multiple adversary models shows that data-flow is necessary to authorize resource retrieval in over 90% of system calls. By making adversary models and the adversary accessibility of all aspects of resource retrieval explicit, we can block resource access attacks system-wide.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Security

Keywords

Resource Access Attacks; Protection

1. INTRODUCTION

Processes need a variety of resources from their operating environment in order to run properly, such as files, IPCs, and sockets. When a process retrieves a resource from the system, it may select any resource to which it is authorized. However, the retrieval of some authorized resources may lead to vulnerabilities, depending on how those resources are to be used. Consider a web server that both serves content supplied by untrusted users and authenticates requests from remote parties. It is easy to see that vulnerabilities would result if the web server used its permissions to read the password file when serving HTML pages or used untrusted HTML pages when authenticating remote parties. In this paper, we explore methods to extend access control mechanisms to prevent the retrieval of resources that will lead to program vulnerabilities, which have been called *resource access attacks* [64].

In principle, such vulnerabilities may occur for a variety of reasons. First, a programmer may not expect that a particular system call invocation may retrieve a resource controlled by one of its adversaries, *expanding the attack surface* of the program [33]. In such cases, the integrity of the running program and/or the data it produces may be compromised by using adversary-controlled input. In the web server example, the use of untrusted content for authentication would compromise the integrity of the web server. Second, a programmer may not expect that a particular system call invocation may retrieve a resource that contains security-sensitive data that should not available to an adversary, leading to a *confused deputy attack* [30]. In such cases, the confidentiality and/or integrity of that sensitive data may be compromised. In the web server example, the serving of the secret password file would compromise the confidentiality of that file. Finally, an adversary may be able to control the inputs that guide the retrieval of resources to redirect a process to a resource of the adversary's choosing, leading to an *attack during name resolution* (e.g., time-of-check-to-time-of-use (TOCTTOU) attacks [42, 8]). By supplying malicious input for file names or controlling the namespace bindings (e.g., links and directories in a file system namespace), adversaries can compromise either the confidentiality or integrity of the process. In the web server example, either an adversary may be able to redirect the server to choose the web content file when the password file is expected or vice versa.

Conventional access control mechanisms are fundamentally unable to prevent such vulnerabilities because permissions are associated with the process at large. *Access control list* mechanisms [37] associate a set of subject identifiers with each resource (i.e., object), authorizing any process running under one of those subject identifiers to access the resource. Thus, any system call invocation by an authorized process will be allowed. Similarly, conventional operating systems that enforce *mandatory access control* [70, 2, 26, 67, 61] (MAC) associate labels with processes and resources, stor-

SACMAT'14, June 25–27, 2014, London, Ontario, Canada.

http://dx.doi.org/10.1145/2613087.2613111.

ing a mapping between the subject labels and the resource labels to which they are authorized. Again, any system call of any process running under a subject label that is authorized to access resources of the target resource's label will be allowed, regardless of how this mapping is stored. In the web server example, access control list mechanisms cannot prevent the password file from accessed by mistake when serving HTML pages, or vice versa, because the web server has access to both resources.

On the other hand, *capability systems* [38] permit processes to restrict the permissions available per system call, but such mechanisms are only available in limited ways in conventional systems. Capability systems permit programmers to select the permissions available per system call, by using specialized references to resources that include permissions, themselves called *capabilities*, and by managing the permissions available to the process flexibly [22]. Researchers have shown that capability systems may be used to prevent some of the vulnerabilities above, such as confused deputy vulnerabilities [30]. However, using a capability system presents a challenge to programmers because they must reason about both the functionality and security of their programs concurrently. As a result, capability systems principles have only been adopted in limited ways, such as sandboxing, that do not permit the flexible control of permissions envisioned for a general capability system [39]. The result is that a variety of ad hoc solutions have been proposed to block resource access attacks (see Related Work in Section 3), but these solutions have been found to be broken [10] or fundamentally flawed [12].

One recent insight is that a system mechanism whose sole purpose is to protect processes during resource retrieval can gather knowledge from both the program and the system to block vulnerabilities [64]. They highlight the fact that access control mechanisms perform two tasks simultaneously, both *protecting* benign processes from attack by limiting adversary access to its resources and *confining* malicious processes from attacking other processes. As a result, access control mechanisms cannot leverage program internal state for their decision-making because a malicious process may spoof the system and break confinement. By separating protection from confinement during resource retrieval in a separate layer of defense called the Process Firewall [64], a variety of resource access attack vulnerabilities can be blocked. As the Process Firewall cannot confine a malicious process, it only augments access control by blocking operations authorized by conventional access control that would only lead to vulnerabilities in the current program state, while still depending on access control for confinement. In the web server example, the Process Firewall examines program internal state to identify whether the system call should retrieve the adversary-accessible HTML file or the adversary-inaccessible password file, and allows only the appropriate resource.

While the Process Firewall mechanism can prevent resource access attacks, the current policy model was also shown to be limited. The Process Firewall allows using program-internal state to determine whether that particular system call should access adversary-accessible or adversary-inaccessible resources. The proposed policy model associates program *entrypoints*, the instructions that invoke the system call library with whether the expected resource retrieved should be adversary-accessible or not. However, the entrypoint may be invoked in multiple contexts, some of which may access different resources, or use name values derived from multiple data flows, only some of which may be under adversary control. As a result, the entrypoint alone is insufficient to express policies to prevent resource access attacks. In addition, the Process Firewall policy model implicitly depends on the definition of *adversary accessibility*. However, we find that there are different types of accessibility that matter depending on the context. In some cases, the adversary must have write permission to the relevant resource, but in other cases the adversary only needs read or execute permissions. The lack of precision and explicit models of adversary accessibility will likely to errors in preventing resource access attacks.

In this paper, we present a novel policy model for expressing rules to protect processes from resource access attacks, such as could be enforced by a Process Firewall mechanism. The policy model is described by a new authorization query rule, called the *resource retrieval query* (RRQ), that binds all of the relevant facets of resource retrieval into a single concise query. We show that the RRQ enables prevention of the resource access attacks listed above in a straightforward way. The main challenges in applying an RRQ policy model are to make adversary accessibility explicit and determine adversary control of names used in resource retrieval. First, we define a novel, role-based approach for describing *adversarial roles*, which list the permissions that dictate whether one subject is an adversary of another. Second, we show how existing work can be applied to compute the data flows impacting a name value used in resource retrieval. Using these methods, adversary control of names can be expressed using concise descriptions, even utilizing entrypoints in some cases, which we call *name control*.

In this paper, we make the following contributions:

- We define a policy model for preventing resource access attacks, whose schema is defined by the *resource retrieval query* (RRQ). In addition to the standard (subject, object, operation), the RRQ requires an explicit representation of the adversary model and program flow to protect the process.

- We define concepts for specifying adversary models and program flows, called *adversarial roles* and *name control*. The former associates subjects with the permissions that define their adversaries, analogously to a role. The latter specifies the control- or data-flow impact on the name used in resource retrieval.

- We evaluate the impact of different adversary models on whether data-flow analysis is necessary to determine adversary accessibility to names. We find that over 60% of the entrypoints in which system calls are invoked can be associated with either adversary-accessible or adversary-inaccessible resources, but over 90% of the system call invocations occur on entrypoints where such a judgement is not possible. Thus, data-flow is important to preventing resource access attacks.

The remainder of the paper is structured as follows. In Section 2, we define the problem of resource access attacks. In Section 3, we examine a variety of efforts to prevent resource access attacks to date and the reasons for their failures. In Section 4, we provide an overview for our approach. In Section 5, we design a policy model based on the *resource*

retrieval query (RRQ) to block resource access attacks and address the challenges in deploying such a model. In Section 6, we evaluate the need for adversary models and dataflow tracking mandated by the RRQ design. In Section 7, we conclude the paper.

2. RESOURCE ACCESS ATTACKS

Once started, a process often needs additional system resources to execute correctly (e.g., libraries, configuration files, logs, etc.) and may need to retrieve task-specific resources to complete any task (e.g., web content files, web requests via sockets, IPCs to worker processes, etc.). We use the term *resource* for objects obtained from the operating system. For convenience, resources are often retrieved by name, using a method known as *namespace resolution* [47, 23]. In a namespace resolution, a client (the process) provides a *name* to a name server (the operating system), which retrieves a reference to the resource to which the name maps via *namespace bindings* managed by the name server.

Resource access attacks are possible because the names, namespace bindings, and resources themselves used by the resolution mechanism may be controlled by adversaries. We use the code snippet in Figure 1 to demonstrate the possible problems. First, many processes obtain resources using names supplied by potential adversaries, particularly server processes that process client requests. In Figure 1, the function `set_up_socket_dir` uses an environment variable `SOCKET_DIR` to name the directory to be created. If adversaries can assign `SOCKET_DIR`, then they could escalate adversary privilege (i.e., by creating a directory in a location they are not authorized for) or could control security-critical operations (i.e., by creating the directory in a location that is accessible to the adversary). These are examples of *untrusted search path* attacks [18], but other kinds of attacks such *directory traversal* and *untrusted library load* similarly occur because adversaries can manipulate the names used in name resolution.

Second, many namespaces allow untrusted parties to specify the namespace bindings used for resolution. Namespaces are often designed to enable sharing of resources among subjects to provide flexibility in application deployment, but such sharing may lead to vulnerabilities if used incorrectly. For example, the X11 script shown in Figure 1 also creates a directory of the name `SOCKET_DIR` in `/tmp/.X11-unix`, where `/tmp` is shared among all processes. Lines 6-8 check if a file already exists that is not a directory, and, if so, moves it to create a fresh directory. In Line 9, the programmer creates the directory, and assumes it will succeed because the previous code had just moved any file that might exist. However, because `/tmp` is a shared directory, an adversary scheduled in between the moving of the file and the `mkdir` might again create a file at `/tmp/.X11-unix`, thus breaking the programmer's expectation. If the file is a link pointing to, for example, `/etc/shadow`, the `chmod` on Line 11 will make it world-readable. In general, using adversary-controlled namespace bindings may lead to problems because adversaries may create bindings that refer to resources they cannot normally access (e.g., symbolic links to attempt *link traversal attacks* [19]). These problems are all difficult to prevent because adversaries may change namespace bindings at any time (e.g., to create race conditions in *TOCTTOU attacks* [42, 8], as in this case).

Third, adversaries may take advantage of the program's ignorance of the namespace and their adversaries' access to that namespace to launch attacks. For example, in the attack above, the adversary plants a link at a file name that the adversary knows that the program will use, `/tmp/.X11-unix`. However, an adversary may cause vulnerabilities simply by creating resources of predictable names in advance (e.g., *squatting attacks* [16]). Such resources are under the adversary, but the victim process may use them without this knowledge, enabling the adversary to control the victim process. In Figure 1, the file created right before the `mkdir` operation was simply a directory the adversary created, the adversary could change the content.

As a result, in order to detect attacks during resource retrieval, any comprehensive defense must authorize the combination of name, bindings, and resource accessed. Current defenses only authorize a subset of such items. Defenses for controlling adversary access to names is limited to ad hoc filtering, which may be error-prone [4]. Most efforts to block attacks during resource retrieval focus on preventing time-of-check-to-time-of-use (TOCTTOU) attacks. Some methods enforce invariants on the resources accessed [17, 56, 60, 52, 68, 59], some enforce use of namespace bindings [13, 51], and some aim for "safe" access methods [20, 58]. Interestingly, the methods are also distinguished by those that augment the program [17, 52, 20, 58] and those that extend the kernel [56, 68, 13, 51, 59, 60]. However, both program and system methods have been found to be fundamentally flawed [10, 12]. Program defenses cannot control how the system allows adversaries to update namespaces and system defenses lack information about programmer intent about which resources are expected in any system call. Our goal in this paper is to address these two limitations by extending access control to reason about program and system concepts.

3. RELATED WORK

In this section, we examine the reasons that current access control models fail to prevent attacks during resource retrieval. We first examine conventional access control and then investigate some proposed research access control models that enable the evolution of permissions as the process runs.

3.1 Limits of Conventional Access Control

A question is whether conventional access control mechanisms may be sufficient to prevent resource access attacks. Several commodity operating systems now enforce mandatory access control policies [70, 67, 2, 43], but these mechanisms cannot prevent such attacks because they grant the same permissions to all system calls for the same process. An important characteristic of resource access attacks is that a resource that is unsafe for a particular victim system call is safe for some other victim system call. For example, a webserver can access `/etc/passwd` legitimately when it wants to authenticate clients, but it should not do so when serving a user web page. As a result, other access control mechanisms that restrict permissions process-wide cannot prevent resource access attacks, such as sandboxes [27, 3, 28, 5].

Capability systems implement an alternative access control mechanism to those above [38], where the programmer chooses the capabilities to present to the operating system to confine access. It has been shown that capability systems can defeat *confused deputy* attacks [30], of which some

```
01 SOCKET_DIR=/tmp/.X11-unix
...
02 set_up_socket_dir () {
03  if [ "$VERBOSE" != no ]; then
04    log_begin_msg "Setting up X server socket directory"
05  fi
06  if [ -e $SOCKET_DIR ] && [ ! -d $SOCKET_DIR ]; then
07    mv $SOCKET_DIR $SOCKET_DIR.$$
08  fi
09  mkdir -p $SOCKET_DIR
10  chown root:root $SOCKET_DIR
11  chmod 1777 $SOCKET_DIR
12  do_restorecon $SOCKET_DIR
13  [ "$VERBOSE" != no ] && log_end_msg 0 || return 0
14 }
```

(a)

```
01 SOCKET_DIR=/tmp/.X11-unix
...
02 set_up_socket_dir () {
03  if [ "$VERBOSE" != no ]; then
04    log_begin_msg "Setting up X server socket directory"
05  fi
06  if [ -e $SOCKET_DIR ]; then
07    mv $SOCKET_DIR $SOCKET_DIR.$$
08  fi
09  mkdir $SOCKET_DIR
10  if [ $? -ne 0 ]; then
11    echo "Unable to create $SOCKET_DIR, possible race!"
12    exit 1
13  fi
14  chmod 1777 $SOCKET_DIR
15  do_restorecon $SOCKET_DIR
16  [ "$VERBOSE" != no ] && log_end_msg 0 || return 0
17 }
```

(b)

Figure 1: A code snippet that is vulnerable to resource access attacks that we found in an X11 startup Bash script in the Ubuntu 11.10 distribution (a), and a possible fix to the TOCTTOU vulnerability (b).

resource access attacks are instances. Some interesting research capability systems have been proposed recently, such as DIFC systems Flume and HiStar [36, 73], which enable flexible control of the permissions used by a process. The problem with capability systems in general is that they push the problem of access control back onto the programmers, presenting yet another API for them to solve the complex problems above. To reduce the complexity on programmers, other recent research on capability-like systems, such as Capsicum [66], relinquishes the flexibility necessary to control access per system call. While we may yet produce an API for programmers to manage capabilities effectively, we propose instead to protect programs from resource access attacks given the current system call API.

3.2 Limits of Research Models

Researchers have previously identified that some attacks may not be prevented unless the access control mechanism accounts for the context in which the program is run. For example, the Brewer-Nash model (aka Chinese Wall model) reduces the permissions a process based on its authorized accesses [11] (e.g., to prevent a conflict of interest). Alternatively, the low-water-mark policy (LOMAC) reduces the permissions of a subject when a lower integrity resource is retrieved [25] (e.g., to prevent unauthorized modification). A corresponding high-water-mark policy blocks raises the secrecy level of subjects when reading higher secrecy data to prevent leakage [69] and other policies protect both secrecy and integrity dynamically [41]. The main limitation of these approaches is that they still restrict the process as a single unit, restricting *all* future accesses.

Researchers have also explored methods for computing permissions based on temporal or contextual properties of the process. For example, traditional role-based access control (RBAC) models [1] were augmented with temporal constraints that alter the permissions available to their processes [6, 35]. In general, a user may be assigned a set of roles, but the roles that may be active at any time may depend on the temporal constraints that have been satisfied. A limitation is that temporal RBAC models apply to all processes simultaneously, whereas resource access attacks affect one process at a time. Alternatively, RBAC models have also been extended to integrate other contextual factors, such as trust, in models that are said to perform *usage control* [53, 54] (UCON). In this model, subjects and objects are associated with attributes, some of which may be mutable based on the subject's access to objects. Authorization requirements are checked prior to and throughout a transaction, which could address attacks that rely on the lack of atomicity, such as TOCTTOU attacks. However, as is typical of conventional access control, UCON (and temporal RBAC models as well) guess at the programmer intent using factors external to the program execution, such as time and system events. In addition, UCON does not reason about how the permissions of other subjects (one's adversaries) may impact whether a resource access should be authorized.

Researchers have also explored methods to reason about permissions by using the program's control or data flow. For example, stack introspection uses the principals responsible for each function on a call stack to deduce the permissions for an operation, such as a resource retrieval [29, 65]. Such methods reason about the security labels of code, but to prevent resource access attacks one must reason about other factors: the data (i.e., used to build names), namespace bindings, or system resources that may be under adversary-control. Alternatively, researchers have developed methods to control access using the program's data flow, enforcing information flow [21, 45]. These methods enable fine-grained reasoning about information flow, which is often difficult for programmers to get right [31], whereas our focus is only on the construction of names for retrieving resources. In addition, these methods do not reason about the implications of bindings on resource retrieval.

3.3 An Alternative: The Process Firewall

Recent work proposed that resource access attacks could be prevented by detecting whether the bindings used and resource accessed in name resolution are unsafe for the "program context" at the time of a name resolution system call, enforced by a kernel mechanism called the Process Firewall [64]. Unsafe accesses were detected using the "adversary accessibility" of the bindings used and resource retrieved in name resolution. For a program context that expected an adversary-accessible resource (e.g., HTML file), the Process Firewall prevents retrieval of resources that are not accessible to program adversaries blocking confused deputy attacks [30]. For a program context that expected an adversary-inaccessible resource (e.g., password file), the Process Fire-

wall prevents retrieval of resources that are accessible to program adversaries limiting the program attack surface [33]. The Process Firewall could also prevent TOCTTOU attacks by restricting multiple program contexts to the same resource.

The Process Firewall is an extension to the SELinux Linux Security Module, which compares resource access system calls authorized by SELinux using a modified version of iptables [40]. The Process Firewall is capable of providing different protections for each system call invocation based on the rules that apply for the current process state (e.g., its call stack), the prior system calls that have been executed by the process, and the current state of the system resource namespace. Because the Process Firewall controls the system resources that individual system call invocations may access, the Process Firewall is analogous to a firewall for the system call interface. Even though the system call interface is much lower latency than the network interface, the Process Firewall incurs only a 2-4% overhead for a variety of macrobenchmarks while enforcing a rulebase of over 1000 rules.

While Process Firewall is capable of preventing resource access attacks efficiently, there is a significant challenge in policy modeling. First, the Process Firewall requires a precise definition of "process context," but this is not a concept in with a precise meaning in general. Cai *et al.* [12] state that a system defense must understand the "programmer intent" to correctly block resource access attacks, but that information is not available. The program "entrypoint"[1] for Process Firewall policies, where runtime analysis was used to classify the entrypoints that always accessed adversary-accessible and adversary-inaccessible resources. However, the Process Firewall experiments exhibited some false positives where entrypoints may be misclassified. Also, there may be many entrypoints that retrieve resources both accessible and inaccessible to adversaries. Second, the Process Firewall requires a precise definition of "adversary accessibility", but there is not a consensus for this concept's meaning. Researchers often apply some notion of a threat model, identifying those resources controlled (e.g., modifiable) by an adversary, but each experiment may propose its own threat model. Even where threat models are computed automatically, different methods are proposed. For example, researchers have proposed multiple automated approaches for computing the resources that are untrusted by each system subject from available discretionary and mandatory access control policies [55, 13, 34, 62].

4. SOLUTION OVERVIEW

To build a policy model for preventing resource access attacks, we want to identify the fundamental principles behind the enforcement policies of the Process Firewall, generalize those principles, and finally simplify the articulation of policy decisions over those principles. To do this, we examine the novel perspective of the Process Firewall design is shown in Figure 2. First, unlike information flow security models, such as Biba integrity [7], the Process Firewall allows processes to retrieve resources controlled by their adversaries, without impacting the permissions of the process overall, unlike LOMAC [25] and other dynamic models [41]. Second, unlike capability systems [38] and recent work on decentralized information flow control [36, 73], the Process Firewall enables enforcement of different policies for individual system call invocations, but it does not require program modifications for such enforcement. Instead, the Process Firewall protects processes during resource retrieval by introspecting into both the system to determine "adversary accessibility" and the process to estimate the "program context" for enforcing rules that deny unsafe retrievals. The Process Firewall authors show that introspection can be used to *protect* the process because even if a malicious process tampers with such information to spoof the Process Firewall, conventional access control is still able to *confine* the process using the original permissions.

As a result, Process Firewall policies must articulate both the "process context" and "adversary accessibility" over each system call invocation. To reason about adversary accessibility and process context, the Process Firewall proposed extending the traditional authorization query to the rule format below:

`pf_invariant(subject, `**`entrypoint`**`, `**`resource_ID`**`, object, `**`adversary_access`**`, operation)` $\rightarrow$ `Y|N|log`

Note that the subject, object, and operation correspond to the normal inputs to an authorization query. In addition, `pf_invariant` extends this query with the following arguments: (1) the *entrypoint*, which is the program instruction that invokes the system call library for this system call invocation; (2) the *resource ID*, which restricts the resource to a specific ID to prevent TOCTTOU attacks [42, 8]; and (3) *adversary access*, which is whether the binding or resource is accessible to process adversaries or not[2]. The entrypoint and resource ID specify requirements on the process context and the adversary access specifies requirements on bindings and the retrieved resource for authorization or denial of the requested operation.

However, as noted in the previous section, while this language for Process Firewall policies enabled the prevention of a variety of resource access attacks, it is prone to both false positives and false negatives. The ad hoc nature of this policy language is the contributing factor to both problems. This languages lacks the generality to express either the program context or adversary accessibility sufficiently to block attacks. However, a concern is that once a general policy language is identified it may be far too complex for policy writers or, importantly in this case, automated tools to produce policies. Fortunately, we have identified some insights, which we highlight below, that motivate our design of policy model that enables prevention of resource access attacks broadly, where the policies may be simplified in many cases.

Expressing the process context requires identifying whether the particular name resolution is expected to retrieve an adversary-accessible resource or not. The Process Firewall paper used only the entrypoint to express the process context for each system call invocation. This limitation led to

[1]A program entrypoint was said to be an instruction that invoked the system call library.

[2]The input *syscall trace* has been removed from this discussion for simplicity, but was part of TOCTTOU defenses. We capture all the information necessary to prevent TOCTTOU attacks by logging resource IDs in the "check" operation and validating them in the "use" operation. We describe this in detail in Section 5.2.

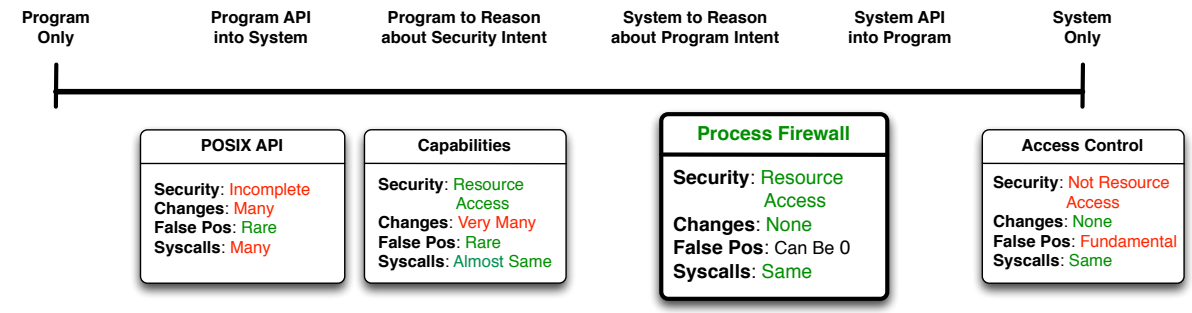

Figure 2: Resource access protection continuum. Until recently no system enforcement mechanisms introspected into the process to protect it from vulnerabilities.

two causes for false positives: different control flow and adversary control of data flow. In one case, a library (that performs system calls) is invoked from multiple callers, but only occasionally does a caller provide a name for an adversary-accessible file. In the other case, the dynamic analysis only tested cases where a particular program entrypoint retrieved files protected from adversaries, but in some rare cases that entrypoint uses adversary input to retrieve files accessible to adversaries. We suggest that the adversary-accessibility of the resource the program intends to retrieve in each system call invocation depends on whether the adversary controls the name of the resource. This is the "process context" that we need to determine to decide whether a retrieved resource is unsafe. In general, determining adversary control of a variable is a data flow problem [21]. However, we find that for many entrypoints, all the data flows leading to that entrypoint are either adversary controlled or not, meaning that we need not track the data flow at runtime in these cases. However we wish to detect whether a name is controlled by an adversary or not, the Process Firewall needs to reason about *adversary control of names*.

Expressing adversary accessibility is conceptually simpler in general, but harder to automate in practice. Conceptually, if an adversary is authorized to perform the requested operation on the resource retrieved, then the resource is adversary-accessible. However, it would be a tedious task to identify the individual resources that may be accessible to adversaries. Researchers have found that the resources accessible to adversaries can be computed from the set of adversaries and the access control policy [55, 13, 34, 62], so we focus on a policy model that expresses adversaries. However, as we mentioned in the last section, there is no agreed-upon method for selecting the adversaries of each process, and selecting them individually would also be tedious. Instead, leverage the notion that researchers had a high-level principle behind their choice of adversaries, so we aim for the Process Firewall to reason about the *the method for identifying adversaries* rather than the individual adversaries.

5. POLICY MODEL DESIGN

In this section, we generalize the authorization rule proposed originally for the Process Firewall [64] to enable more accurate control during resource retrieval with less likelihood of false positives. In Section 5.1, we propose a new authorization rule, called a *resource retrieval query* (RRQ), and show how it can prevent the various types of resource access attacks in Section 5.2. In Section 5.4, we show how to express adversary control of names in terms of *flow statements*. Finally, in Section 5.3, we show how to express the adversaries of a process using *adversarial roles*.

5.1 Resource Retrieval Queries

The goal is to block processes from retrieving resources that can only lead to resource access attacks. We find that in general we need to know: (1) whether the name resolution process is under adversary control and (2) whether the retrieved resource is adversary accessible or not.

To understand exactly what we mean it is necessary to have precise definitions for adversary control and adversary accessibility. In general, adversaries may *control* the inputs to a name resolution, names and bindings, and *be authorized to access* the output of name resolution, the resource retrieved. For inputs, a name or binding is *adversary-controlled* if the adversary is authorized to modify the source of that input. For names, if an adversary can modify any of the resources from which the name is constructed, then the adversary is said to control that input. For bindings, if an adversary can modify the filesystem links (e.g., directories and symbolic links) used in a name resolution, then the adversary is said to control the binding. For outputs, a resource is *adversary-accessible* if the adversary is authorized to perform the requested operation on the retrieved resource. If an adversary is only authorized to read a resource, then a system call that retrieves a resource for writing would be considered inaccessible to that adversary.

On every system call that performs name resolution, we propose to query both conventional access control (for confinement and some protection) and perform a second query designed to prevent resource access attacks, called a *resource retrieval query* (RRQ). We define the format of the RRQ below.

$$\texttt{RRQ}(\texttt{subject}, \textbf{adversarial\_role}, \texttt{object}, \textbf{name\_control}, \textbf{object\_control}, \texttt{operation}) \rightarrow \texttt{Y}|\texttt{N}|\texttt{log}$$

Like `pf_invariant`, the RRQ specifies the conditions under which a resource retrieval is deemed unsafe, where the default assumption is that an operation is safe (i.e., after all it has been authorized by the traditional access control policy). Note that this semantics is analogous to that of a traditional network firewall, where the default is to allow the traffic.

Our definition of RRQ makes the following changes relative to `pf_invariant`. First, the RRQ associates each subject with its *adversarial role*, which explicitly identifies the threat model used for identifying this subject's adversaries. That is, the subject's adversarial role explicitly relates the system call's subject to the other system subjects who may threaten it by identifying the *permission sets accessible to non-adversaries*. The adversarial role is used to identify adversary control of names and bindings and adversary accessibility of resources. Second, we express adversary control of names using *name control*, which associates a control or data flow of the program with the authorized bindings and resources for that flow (via object control below). Note that the entrypoint is one instance of a name control value, as it implies a single node flow graph. Importantly, we show that in some cases complex data or control-flow relationships may be reduced to a simpler representation, even down to an entrypoint alone, without loss of information. Finally, we change the argument *adversary_accessibility* from `pf_invariant` into *object control*. Object control simply specifies a constraint on the authorized resources given a name control's flow. Normally, object control will either specify that adversary-accessible or adversary-inaccessible resources may be authorized given a name control flow. Note that the object being authorized is either a binding or a resource[3], where the operation being performed uniquely identifies whether the object is a binding or a resource. We note that the *resource ID* field of `pf_invariant` is folded into the object control by expressing further constraints on the objects that may be accessible given a flow and optionally prior resources retrieved (e.g., to prevent TOCTTOU attacks).

When an RRQ query is run, the enforcement mechanism (e.g., Process Firewall) uses the subject and adversarial role to determine whether the object (binding or resource) being authorized is unsafe and must be blocked. First, the enforcement mechanism will determine whether a name control rule matches the control (e.g., call stack) or data flow (e.g., variable taints) that the program used to retrieve the resource for that system call invocation. If so, then the enforcement mechanism will compare the object control value to the RRQ's object (i.e., bindings used or resource retrieved depending on the operation) to determine whether the object matches the object control requirement. If so, then the rule's action is taken (accept, deny, etc.).

5.2 Preventing Attacks with RRQs

In this section, we show that confused deputy, expanded attack surface, and TOCTTOU attacks can be blocked using RRQs.

For a confused deputy attack [30], the problem is that an adversary in control of a name and/or binding can redirect the victim to a resource that is not accessible to the adversary. First, consider the case where the name is controlled by an adversary. Using RRQs, we identify the name control flows where the name is controlled by an adversary. If such a flow matches, then an adversary-inaccessible resource would be unsafe, so the RRQ object control would specify "adversary accessible" for a "deny" rule. Note evaluating adversary control of bindings is not necessary for the decision.

[3]An operating system will perform an authorization for each binding in the pathname and the final resource separately. Thus, each authorization will evaluate access to either a binding or a resource.

Second, consider the case where the name is not controlled by an adversary, but the name resolution uses an adversary-controlled binding. Normally, we would expect to retrieve an adversary-inaccessible resource, but the the use of an adversary-controlled binding to retrieve such a resource may enable a victim to be redirected to an unexpected resource. Using RRQs, the object control would specify "adversary accessible" bindings for a "deny" rule. This defense was implemented as a library function called "safe-open" by Chari *et al.* [13].

For an expanding attack surface attack [33], the problem is that an adversary is in control of a resource, but not the name. Using RRQs, we identify the name control flows when the adversary does not control the name and associate those with the object control where the resource is "adversary accessible" for a "deny" rule. Such a rule would be in used in combination with the "safe-open" RRQ above to prevent multiple possible attacks in one system call invocation.

Finally, TOCTTOU attacks occur because an adversary can change the bindings used in name resolution to direct the victim to a different resource, even when the same name is used. Using RRQs, we identify a name control flow that uses a name that will reused. An RRQ can log the resource in an object control specification of the rule when the log directive is provided. Other RRQs define the name control flows when that object control will be enforced. In these RRQs, the object control will be the resource logged previously.

The power of the RRQ approach lies in the adversarial role and name control, which are not specified in the examples above. The adversarial role enables precise specification of the meaning of adversary controlled and adversary accessible relative to a role (i.e., set of permissions). As we can see above, knowledge about adversary control and adversary accessibility is fundamental to preventing resource access attacks. In addition, the name control enables precise specification of adversary control of names, which was missing in the Process Firewall's rule language. As we can see above, knowledge of the adversary's control of names is fundamental to reasoning about resource access attacks. We examine how to use both of these concepts in the following subsections.

5.3 Defining Adversaries

To compute adversary control and adversary accessibility, we first need to identify the subjects who are adversaries. Given a set of adversarial subjects and an access control policy, researchers have shown that they can compute the permissions (resources and operations) that enable adversary control and accessibility [55, 13, 34, 62].

To help guide our thinking, we first review how researchers have applied adversary accessibility in experiments previously. For example, researchers have long used user IDs in discretionary access control systems to distinguish friend from foe [55, 13]. In these experiments, all processes trust root processes and those running under the same user ID. If processes running under other user IDs (except root) may modify the input used to build names, the bindings used in name resolution, and/or the end resource retrieved, then caution must be observed during resource retrieval. Similarly, researchers have applied a variant of this idea to mandatory access control (MAC) policies, using the subject labels as the guide [34]. Since MAC policies aim to reduce the privileges of root processes, a problem is that it is more difficult

to identify which subject labels that run root code are trustworthy. Researchers have proposed a threat model whereby processes running under a particular subject label only trust processes that have permissions to modify their executable code or write to kernel memory [62] (applied transitively). Interestingly, this threat model was found to correspond well to reported vulnerabilities. That is, these relatively simple threat models have proven useful for experimentation.

A problem has been that researchers have not been convinced that these simple threat models used in experiments can be used in practice to prevent attacks without causing numerous false positives. The problem has been that just identifying where an adversary has access to a binding or a resource is insufficient to detect an unsafe access. Consider the confused deputy attack [30]. If the process uses adversary-inaccessible bindings to retrieve an adversary-inaccessible resource, it may still be unsafe if the adversary controls the name used. Thus, adversary accessibility of system objects (bindings and resources) must be combined with adversary accessibility of program objects (names) to reason about resource access attacks comprehensively. Thus, the combination of system and program is required to leverage adversary accessibility effectively.

A second problem is that administrators did not have any guidance for selecting adversaries for a program. However, as prior research has shown there are a few simple and effective ways of describing the adversaries of a program. In this paper, we highlight three such cases, which we will express below using adversarial roles. First, researchers have often employed the notion that processes running with your user ID and a root user ID are trusted, which we call the *root role*, such as used previously in DAC systems [55, 13]. This approach reflects some practical issues in trust. All processes must trust root processes, and any process with the same user ID can perform the same operations. Second, researchers recently proposed an approach for identifying trusted subjects in MAC policies, where one subject only trusts the MAC subjects that may write to kernel memory or their executable code [62], which we call the *local role*. The advantage of this approach is that it also considers the practical issues of trust. Any process that can modify kernel memory or your processes's executables must be trusted. Third, researchers have proposed adversaries directly as those processes with network access, which we call the *remote role*. Some conventional operating systems employ a policy where only network-facing daemons are untrusted [49, 50]. This approach differs from the ones above in that it focuses on adversaries rather than trusted processes, although the number of adversaries is smaller in practice.

In formulating an approach to express adversarial roles we highlight three facts. First, whether a subject is an adversary does depend on the permissions they hold. Second, some permissions may indicate that the subject is trusted whereas others identify the subject as an adversary. Third, some of these permissions are subject-specific, such as the permissions to modify the subject's executable code files. As a result, we define an *adversarial role* relative to a subject s as $AR(s) = (P, f(s, \mathcal{P}), \{\sqcup, \dashv\})$, where P is a set of permissions, $f(s, \mathcal{P})$ is a function that computes subject-specific permissions from the access control policy $\mathcal{P}$, and $\{t, a\}$ identifies whether possession or lack of those permissions makes one an adversary. For example, local role above includes a set of permissions to write to kernel objects in P and subject-specific permissions to their executable code files $f(s, \mathcal{P})$[4], where the subjects with those permissions are the only trusted subjects $\{t\}$. As in role-based access control, administrators could predefine such roles. In practice, we expect that the number of adversarial roles in use will be modest, enabling such adversarial role definitions to be reused across deployments, as roles can be reused for multiple users.

5.4 Expressing Name Control

In general, whether a resource retrieval is unsafe or not requires knowledge about the adversary's control of the names used in resource retrieval. Our goal is to define a method for specifying whether a name is adversary-controlled or not using the program constructs. However, as described above, program entrypoint is too limited to express adversary control of names in all cases. Nonetheless, where an entrypoint is sufficient to express adversary accessibility, we would like our specification to "compile" into an entrypoint.

Whether a name may be modified by an adversary is fundamentally a data flow problem. For example, Denning defines an information flow model for programs [21] that is sufficient to identify whether the value of a name variable is controlled by an adversary of a program statically. This model has been applied to develop automated methods [45] to determine whether the security requirements of a "channel" in which a variable is used (e.g., a resource retrieval system call) complies with the security requirements of the variable itself (e.g., an adversary-controlled name).

In this context, we could apply static program information flow analysis [45] to determine whether all the data flows to a name variable at a system call entrypoint include data from an adversary controlled input (e.g., file). Conversely, we could use static program analysis to determine whether no data flow to a name variable at a system call entrypoint includes data from an adversary controlled input. In either case, then knowledge of the entrypoint alone is sufficient to identify whether the name is adversary-controlled or not, enabling enforcement using the Process Firewall rules in Section 4. Thus, where static analysis can prove the adversary control of an resource retrieval entrypoint in all cases, the entrypoint is sufficient for RRQ rules as well.

However, not all information flows may be derived statically, as some names may only be under adversary control some of the time. For example, some names may be derived from inputs that may sometimes originate from adversary-controlled files and sometimes not. In this case, program information flow analysis utilizes runtime labels [44]. Such analysis would log the runtime labels of files that may be used to provide name input and determine the label of the name variables at name resolution system calls. This approach has been leveraged to connect program information flow with system labels for SELinux [32]. Using this method, adversary control of names is determined at runtime, which the enforcement mechanism can then extract to enforce the RRQs. For the prior work, APIs were designed to enable information flow-aware programs (in the Jif language [46])

[4]In an SELinux policy [57], executable files have a special label that distinguishes them from regular files. The SELinux policy also specifies the subject labels that may execute those files. If this subject can execute a file of a particular object label, then permission to modify that file (label) is added to the adversarial role [62].

to retrieve the necessary labeling information from SELinux to authorize access (in the traditional manner). As shown in Figure 2, the Process Firewall works in the opposite direction by extracting program context (the label of the name variable) from the program memory. However, this can easily be done by creating a map in program memory between the name variable at the entrypoint and the label of the variable. Thus, the RRQ name controls associate the variable, entrypoint and adversary-control status expected to dictate the allowed adversary accessibility.

A practical problem is that very few programs are written in the Jif programming language. In practice, dynamic taint analysis [71, 24] would be used to determine whether an adversary-controlled a name variable. However, full dynamic taint analysis is very expensive. Fortunately, we are only interested in a few variables, but unfortunately, many complex data flows may impact their values. Dynamic taint analyses have been designed to distinguish between different (adversary-controlled and not) sources [9, 48, 71]. However, taint analysis does not handle certain kinds of information flow (implicit flows), meaning that it is currently less accurate than the information flow analysis above. Nonetheless, using dynamic taint analysis techniques, we can write the same RRQ rules.

Finally, note that programmers may want to use application-specific sanitization to untaint a flow that has a dependence on adversary input. Information flow analysis uses endorsers to remove adversary control from data, whereas taint analyses also support this in a somewhat more ad hoc way. Note that with above analyses, RRQs could be generated automatically, excepting for such sanitization operations.

6. EVALUATION

In this section, we evaluate the impact of different adversary models on the use of the entrypoint alone to describe name controls. This study was performed on a newly-installed Ubuntu 12.04 Desktop filesystem protected by the SELinux reference policy [57]. The analysis data was produced using a runtime analysis driven by Linux package test suites and normal use. As runtime analysis is inherently incomplete, this analysis provides an upper bound for the number of entrypoints that may be classified as accessing only one type of resource.

Under that limitation, we notice three interesting trends. First, as shown in Table 1, the adversary model has a significant impact on the ability to classify entrypoints. Second, in all models, a significant number of entrypoints are classified as either retrieving only adversary-accessible or adversary-inaccessible resources. In those cases, RRQs can use the entrypoint for name control. However, Table 2 shows that a vast majority of the individual system call invocations are made at entrypoints that retrieve both adversary-accessible and adversary-inaccessible resources. This shows that it is important that RRQs handle data flow in a more flexible way for the remaining entrypoints, as they are most often used.

Adversary Models. We evaluate three different adversary models: one based on the DAC policy, and two based on SELinux MAC policies. All of them assume both remote and local adversaries. In the DAC adversary model (**Root** from Section 5.3), a user ID has as adversaries all other user IDs, excepting the superuser `root`. This model holds for systems that use only DAC to control access. The first MAC adversary model [62] (**Local** from Section 5.3) assumes only a minimal system and application trusted computing base. This model is conservative; for example, network daemons are adversarial to the local system. This scenario holds when network daemons are broken into, and try to further escalate privileges. The second MAC adversary model (**User**) assumes only two subjects – `user_t` and `guest_t` (assigned to unprivileged users) – and all the resources modifiable by these two subjects, untrusted. This corresponds to adversarial local users who have a login to the system and are constrained by MAC policies.

Adversary Model	Adv. Acc Resources	Adv. Inacc Resources	Both
Root	8334	360	2371
Local	5436	1675	3954
User	8652	880	1533

Table 1: Adversary accessibility for entrypoints.

Defense Rule Invoked	Syscalls	%
Only adv. inacc.	**Root** 439379	9.40%
	Local 29017	0.62%
	User 74716	1.6%
Only adv. acc.	**Root** 582	0.01%
	Local 2035	0.04%
	User 1073	0.02%
safe_open	**Root** 138825	3.0%
	Local 1019481	21.8%
	User 119560	2.5%
Total	4671037	–

Table 2: The number and percentage of system calls for which the entrypoint alone is sufficient to prevent resource access attacks.

Sufficiency of Entrypoint Context. In Table 1, we show the number of entrypoints that retrieve only adversary-accessible resources, only adversary-inaccessible resources, or both, for the three different adversary models.

This table shows that under the **Root**, **Local** and **User** models, we can classify 78.5%, 64.2% and 86.1% of entrypoints respectively as accessing either only adversary-accessible resources or only adversary-inaccessible resources. Thus, it appears likely that a significant portion of RRQs can use entrypoints to describe name controls.

System Call Frequency. We examine the distribution of system calls relative to those that can be protected using name controls specified by entrypoint. This tells us the number of system calls, regardless of entrypoint, for which the entrypoint alone is sufficient to describe the program data flow. Table 2 shows the number of times a system call associated with that classification was invoked. In this case, we also include *safe open* defenses which do not require knowledge of the adversary control of names for comparison.

Table 2 shows that only a small percentage of the system calls are run using entrypoints that only access adversary-accessible or adversary-inaccessible resources. On examining the cause for the low percentages, we found that many system calls are made through a few commonly invoked entrypoints, and most of these commonly invoked entrypoints

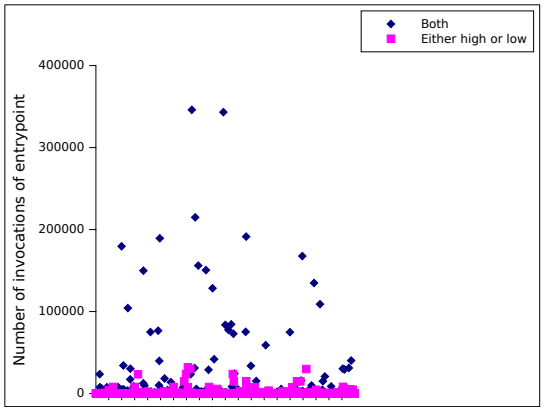

Figure 3: Number of Invocations per Entrypoint

use both adversary-accessible and adversary-inaccessible resources. This is shown in Figure 3. Many of these entrypoints belong to common programs; for example, the 50 most commonly invoked entrypoints belonged to either the Python or Bash interpreter. For these entrypoints, the RRQ's name controls should in addition be based on knowledge of program data flow.

7. CONCLUSIONS

In this paper, we presented a policy model to prevent resource access attacks. While a variety of methods have been proposed, this is the first method to make explicit all aspects of resource retrieval: adversaries, using *adversary roles*; adversary control of the names used in resource retrieval, using *name controls*; the adversary control of the bindings used in name resolution; and adversary access to the resource retrieved. In particular, we focus on the challenges of adversary roles and name control via program flows, defining simple models for expressing each that leverages a variety of prior work. Our evaluation shows both that the adversary models chosen make a significant difference how resource access attacks must be prevented and that data-flow tracking is fundamental to a comprehensive defined for resource access attacks. Often, well over 90% of the individual system calls require data-flow tracking to prevent resource access attacks accurately.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-FA9550-12-1-0166. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

9. REFERENCES

[1] RBAC '98: Proceedings of the Third ACM Workshop on Role-based Access Control, 1998. Chairman-Youman, Charles and Chairman-Jaeger, Trent.

[2] Solaris Trusted Extensions Developer's Guide. `http://docs.sun.com/app/docs/doc/819-7312`, 2008.

[3] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[4] D. Balzarotti *et al.* Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

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

[6] E. Bertino, P. A. Bonatti, and E. Ferrari. Trbac: A temporal role-based access control model. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):191–233, 2001.

[7] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.

[8] M. Bishop, M. Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[9] BitBlaze. BitBlaze binary analysis project. `http://bitblaze.cs.berkeley.edu`, 2014.

[10] N. Borisov *et al.* Fixing races for fun and profit: How to abuse atime. In *USENIX Security '06*, 2005.

[11] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1989.

[12] X. Cai *et al.* . Exploiting Unix File-System Races via Algorithmic Complexity Attacks. In *IEEE SSP '09*, 2009.

[13] S. Chari, S. Halevi, and W. Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS*, 2010.

[14] S. Chari, S. Halevi, and W. Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS*, 2010.

[15] S. Chari *et al.* Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation. In *NDSS '10*, 2010.

[16] E. Chin *et al.* Analyzing Inter-Application Communication in Android. In *MobiSys*, 2011.

[17] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *USENIX Security Symposium*, pages 165–176, 2001.

[18] CWE. CWE-426: Untrusted Search Path. `http://cwe.mitre.org/data/definitions/426.html`.

[19] CWE. CWE-59: Improper Link Resolution Before File Access. `http://cwe.mitre.org/data/definitions/59.html`.

[20] D. Dean and A. J. Hu. Fixing races for fun and profit: How to use access (2). In *USENIX Security Symposium*, pages 195–206, 2004.

[21] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.

[22] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, 1966.

[23] Domain Names - Implementation and Specification. `http://http://www.ietf.org/rfc/rfc1035.txt`.

[24] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 1–6, 2010.

[25] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[26] Mandatory Access Control - FreeBSD. `http://www.freebsd.org/handbook/mac.html`.

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

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

[29] L. Gong, R. Schemers, and S. Microsystems. Implementing protection domains in the java development kit 1.2, 1988.

[30] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[31] B. Hicks, K. Ahmadizadeh, and P. McDaniel. From Languages to Systems: Understanding Practical Application Development in Security-typed Languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, December 2006.

[32] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: building and executing applications that enforce system security. In *USENIX Annual Technical Conference*, June 2007.

[33] M. Howard, J. Pincus, and J. Wing. Measuring Relative Attack Surfaces. In *Proceedings of Workshop on Advanced Developments in Software and Systems Security*, December 2003.

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

[35] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A generalized temporal role-based access control model. *IEEE Trans. on Knowl. and Data Eng.*, 17(1):4–23, Jan. 2005.

[36] M. N. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[37] B. W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[38] H. M. Levy. *Capability-based Computer Systems.* Digital Press, 1984. Available at `http://www.cs.washington.edu/homes/levy/capabook/`.

[39] T. A. Linden. Operating system structures to support security and reliable software. *ACM Computing Surveys*, 8(4):409–445, Dec. 1976.

[40] R. Marmorstein and P. Kearns. A Tool for Automated iptables Firewall Analysis. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[41] D. McIlroy and J. Reeds. Multilevel windows on a single-level terminal. In *Proceedings of the (First) USENIX Security Workshop*, Aug. 1988.

[42] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Syst. J.*, 13:230–252, September 1974.

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

[44] A. C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[45] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[46] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9:410–442, October 2000.

[47] R. Needham. *Chapter: Names. In S. Mullender (Ed): Distributed Systems.* Addison-Wesley, 1989.

[48] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the 2005 Network and Distributed System Security Symposium*, 2005.

[49] AppArmor Linux application security. `http://www.novell.com/linux/security/apparmor/`, 2008.

[50] Security-enhanced linux targeted policy. `http://www.centos.org/docs/5/html/Deployment_Guide-en-US/rhlcommon-chapter-0001.html`.

[51] OpenWall Project - Information security software for open environments. `http://www.openwall.com/`, 2008.

[52] J. Park, G. Lee, S. Lee, and D.-K. Kim. Rps: An extension of reference monitor to prevent race-attacks. In *PCM (1) 04*, 2004.

[53] J. Park and R. Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 57–64, New York, NY, USA, 2002. ACM.

[54] J. Park and R. Sandhu. The UCONABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, Feb. 2004.

[55] C. Pu and J. Wei. A Methodical Defense against TOCTTOU Attacks: The EDGI Approach. In *ISSSE*, 2006.

[56] K. suk Lhee and S. J. Chapin. Detection of file-based race conditions. *Int. J. Inf. Sec.*, 2005.

[57] Reference Policy. `http://oss.tresys.com/projects/refpolicy`, 2008.

[58] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *FAST*, volume 8, pages 1–18, 2008.

[59] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In

Proceedings of the 12th USENIX Security Symposium, pages 243–255, 2003.

[60] P. Uppuluri, U. Joshi, and A. Ray. Preventing race condition attacks on file-systems. In *SAC-05*, 2005.

[61] C. Vance, T. Miller, R. Dekelbaum, and A. Reisse. Security-enhanced darwin: Porting selinux to mac os x. In *Proceedings of the Third Annual Security Enhanced Linux Symposium, Baltimore, MD, USA*, 2007.

[62] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2012)*, May 2012.

[63] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2012)*, May 2012.

[64] H. Vijayakumar, J. Schiffman, and T. Jaeger. Process firewalls: protecting processes during resource access. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 57–70. ACM, 2013.

[65] D. S. Wallach, A. W. Appel, and E. W. Felten. Safkasi: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, Oct. 2000.

[66] R. Watson, J. Anderson, and B. Laurie. Capsicum: practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[67] R. N. M. Watson. TrustedBSD: Adding trusted operating system features to FreeBSD. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 15–28, 2001.

[68] J. Wei *et al.* A methodical defense against TOCTTOU attacks: the EDGI approach. In *IEEE International Symp. on Secure Software Engineering (ISSSE)* , 2006.

[69] C. Weissman. Security controls in the adept-50 time-sharing system. In *Proceedings of the November 18-20, 1969, Fall Joint Computer Conference*, AFIPS '69 (Fall), pages 119–133, New York, NY, USA, 1969. ACM.

[70] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, August 2002.

[71] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[72] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[73] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.