JIGSAW : Protecting Resource Access by Inferring Programmer Expectations

Hayawardh Vijayakumar[1], Xinyang Ge[1], Mathias Payer[2], and Trent Jaeger[1]

[1]SIIS Laboratory, Department of CSE
The Pennsylvania State University
{hvijay,xxg113,tjaeger}@cse.psu.edu
[2]EECS Department
University of California Berkeley
mathias.payer@nebelwelt.net

Abstract

Processes retrieve a variety of resources, such as files, from the operating system to function. However, securely accessing resources has proven to be a challenging task, accounting for 10-15% of vulnerabilities reported each year. Current defenses address only a subset of these vulnerabilities in ad-hoc and incomplete ways. In this paper, we provide a comprehensive defense against vulnerabilities during resource access. First, we identify a fundamental reason that resource access vulnerabilities exist – a mismatch between programmer expectations and the actual environment the program runs in. To address such mismatches, we propose JIGSAW, a system that can automatically derive programmer expectations and enforce it on the deployment. JIGSAW constructs programmer expectations as a *name flow graph*, which represents the data flows from the inputs used to construct file pathnames to the retrieval of system resources using those pathnames. We find that whether a program makes any attempt to filter such flows implies expectations about the threats the programmer expects during resource retrieval, the enabling JIGSAW to enforce those expectations. We evaluated JIGSAW on widely-used programs and found that programmers have many implicit expectations. These mismatches led us to discover two previously-unknown vulnerabilities and a default misconfiguration in the Apache webserver. JIGSAW enforces program expectations for approximately 5% overhead for Apache webservers, thus eliminating vulnerabilities due to resource access efficiently and in a principled manner.

1 Introduction

Processes retrieve a variety of *resources* from the operating system to function. A resource is any abstraction that the system call API of an operating system (OS) offers to a process (apart from a process itself). Examples of resources are files (configuration, data, or log files), network ports, or interprocess communication channels (IPCs) such as sockets and shared memory. Such OS abstractions free the programmer from having to know details of the underlying hardware and allow her to write portable code. Conceptually, resource access is a procedure that takes as input the name (e.g., filename) and namespace bindings (e.g., directories or symbolic links), and returns the resource (e.g., file) as output to the process.

Securely accessing resources has proven to be a challenging task because of adversarial control of the inputs to resource access. Adversaries may control the input name or a binding to direct victim processes to unsafe resources. In the well-known *time-of-check to time-of-use* (TOCTTOU) attack [6], an adversary exploits the non-atomicity between check operations (e.g., access) and use operations (e.g., open) to redirect the victim to resources of the adversary's choosing. Other attacks are link following, untrusted search paths, Trojan-horse library loads, and directory traversal. These attacks are collectively referred to as *resource access attacks* [40]. 10-15% of the vulnerabilities reported each year in the CVE database [12] are due to programs not defending themselves against these attacks.

Current defenses against such vulnerabilities in conventional OSes are ad-hoc and fundamentally limited. First, traditional access control is too coarse-grained to prevent resource access vulnerabilities. A process may have legitimate access to both low-integrity adversarial files and high-integrity library files; however, the resource access to load libraries should not access adversary files (and vice versa). Traditional access control does not differentiate between these resource accesses, and thus cannot protect programs. Second, defenses in the research literature require either system or program modifications. System defenses have been mainly limited to TOCTTOU [11, 13, 25–27, 35–38, 44] and link following [10], or require programs to be written to new APIs [19,29,34,42]. However, system defenses are fund-

mentally limited because they do not have sufficient information about programs [8], and new APIs do not protect existing programs.

This paper presents an approach to automatically protect programs that use the current system call API from resource access vulnerabilities. We make the following observations: first, we find that a fundamental cause for resource access vulnerabilities is programmer expectations not being satisfied during the program's system deployment. For example, during a particular resource access, a programmer might expect to fetch a resource that is inaccessible to an adversary (e.g., a log file in `/var/log`) and thus not add defensive code checks, called *filters*, to protect against adversarial control of names and bindings. However, this expectation may not be consistent with the view of the OS distributors (e.g., Red Hat, Ubuntu) who actually frame the access control policy. Thus, if permissions to `/var/log` allow adversary access (e.g., through a world-writable directory), adversaries can compromise the victim program. Our second insight is that we can *automatically* infer if the programmer expected adversarial control at a particular resource access or not, without requiring any annotations or changes to the program. We do this by detecting the presence of name and binding filters that programmers place in the program.

In this paper, we develop JIGSAW, the first system to provide automated protection for programs during resource access without requiring additional programmer effort. JIGSAW infers programmer expectations and enforces these on the deployment[1]. JIGSAW is based on two conceptually simple invariants – that the system deployment's attack surface be a subset of the programmer's expected attack surface, and that resource accesses not result in confused deputy attacks [16]. These invariants, if correctly evaluated and enforced, can theoretically provide complete program protection during resource access. JIGSAW operates in two phases. In the first phase, it mines programmer expectations by detecting the presence of filters in program code. Using these filters, JIGSAW constructs a novel representation, called the *name flow graph*, from which the programmer's expected attack surface is derived. We show that anomalous cases in the name flow graph can be used to detect vulnerabilities to resource access attacks. In the second phase, JIGSAW enforces the invariants by leveraging the open-source Process Firewall of Vijayakumar *et al.* [40], a Linux kernel module that (i) knows about deployment attack surface using the system's adversary accessibility, and (ii) introspects into the program to identify the resource access and to enforce its expectations as determined in the first phase.

[1]Informally, JIGSAW enables "fitting" the programmer's expectations on to its deployment.

We evaluate our technique by hardening widely-used programs against resource access attacks. Our results show that in general, programmers have many implicit expectations during resource access. For example, in the Apache web server, we found 65% of all resource accesses are implicitly expected not to be under adversary control. However, this is not conveyed to OS distributors in any form, and may result in vulnerabilities. JIGSAW can be use to detect such vulnerabilities, and we did find two *previously-unknown* vulnerabilities and a default misconfiguration. However, the key feature of JIGSAW is that it protects program vulnerabilities during resource access whenever there is a discrepancy between the programmers' inferred expectations and the system configuration, without the need to modify the program or the system's access control policies. We also find that the Process Firewall can enforce such protection to block resource access vulnerabilities whilst allowing legitimate functionality for a modest performance overhead of approximately 5%. An automated analysis as presented in this paper can thus enforce efficient protection for programs during resource access at runtime.

In summary, we make the following contributions.

- We precisely define resource access vulnerabilities and show how they occur due to a mismatch in expectations between the programmer, the OS distributor, and the administrator,
- We propose two invariants that, if evaluated and enforced correctly, can theoretically provide complete program protection during resource access,
- We develop JIGSAW, an automated approach that uses the invariants to protect programs during resource access by inferring programmer expectation using the novel abstraction of a name flow graph, and
- We evaluate our approach on widely-used programs, showing how programmers have many implicit expectations, as demonstrated by our discovery of two previously-unknown vulnerabilities and a default misconfiguration in our deployment of the Apache web server. Further, we show that we can produce rules to enforce these implicit assumptions efficiently using the Process Firewall on any program deployment.

2 Problem definition

In this section, we first give a precise definition of when a resource access is vulnerable. This definition classifies vulnerabilities into two broad categories. We then identify the fundamental cause for each of these vulnerability categories – mismatch between programmer expectation and system deployment, and difficulty in writing proper defensive code.

```
01  conf = open("httpd.conf");
02  log_file = read(conf);
03  socket = bind(port 80);
04  open(log_file, O_CREAT);          // File Squat
05  loop {
06    html_page = recv(socket);
07    strip(html_page, "../");        // Directory Traversal
08    stat(html_page not symlink);    // TOCTTOU Race
09    open(html_page, O_RDONLY);      // TOCTTOU Race, Symlink
10    write(client_socket, "200 OK");
11    log("200 OK to client")
12  }
```

Figure 1: Motivating example of resource access vulnerabilities using a typical processing cycle of a web server.

2.1 Resource Access Attacks

A resource access occurs when a program uses a *name* to resolve a *resource* using namespace *bindings*. That is, the inputs to the resource access are the name and the bindings, and the output is the final resource. Figure 1 shows example webserver code that we use throughout the paper: the webserver starts up and accesses its configuration file (line 2), from which it gets the location of its log file. It then binds a socket on port 80 (line 3), opens the log file (line 4), and waits for client requests. When a client connects, it receives the HTTP request (line 6), uses this name to fetch the HTML file (line 9). Finally, it writes the status code to its log file (line 11).

Let us examine some possible resource access vulnerabilities. Consider line 6. Here, the program receives a HTTP request from the client, and serves the page to the client. The client can supply a name such as `../../etc/passwd`, and if the server does not properly sanitize the name (which it attempts to do in line 7), the client is served the password file on the server. This is a *directory traversal* vulnerability. Next, consider the check the server makes in line 8. Here, the server checks that the HTML file is not a symbolic link. The reason for this is that in many deployments (e.g., a university web server serving student web pages), the web page is controlled by an adversary (i.e., student). The server attempts to prevent a symbolic link vulnerability, where a student links her web page to the password file. However, a race condition between the check in line 8 and the use in line 9, leads to a *link following* vulnerability exploiting a *TOCTTOU race condition*.

To see how such vulnerabilities can be broadly classified, we introduce adversary accessibility to resources and adversarial control of resource access. We then define when resource accesses are vulnerable, and use this to derive a classification for vulnerabilities.

Adversary accessible resources. An *adversary-accessible resource* is one that an adversary has permissions to access (read for secrecy attacks, write for integrity attacks) under the system's access control policy. The complement set is the set of *adversary-inaccessible* resources.

Expected/Safe Resource	*Malicious/Unsafe Resource*	*Vulnerability Class*
Adversary-Inaccessible (Hi) Resource	*Adversary-Accessible (Lo) Resource*	**Unexpected Attack Surface** Untrusted Search Path File/IPC Squat PHP File Inclusion
Adversary-Accessible (Lo) Resource	*Adversary-Inaccessible (Hi) Resource*	**Confused Deputy** Link Following Directory Traversal TOCTTOU races

Table 1: Adversaries control resource access to direct victims to adversary-accessible resources when the victim expected an adversary inaccessible resource and vice-versa.

Adversary control of resource access. An adversary controls the resource access by controlling its inputs (the name or a binding). An adversary controls a binding if she uses her write permissions in a directory to create a binding [39]. An adversary controls a name if there is an explicit data flow[2] from an adversary-accessible resource to the name used in resource access. The adversary needs write permissions to these resources to control names.

The directory traversal vulnerability above relies on the adversary's ability to control the name used in resource access. The link following vulnerability relies on the adversary's ability to control the binding (creating a symbolic link).

Resource Access Vulnerability. A resource access is vulnerable, i.e., a resource access attack is successful, when an adversary uses her control of inputs to resource access (the name or a binding) to direct a victim to an adversary-accessible output resource when the victim expected an adversary-inaccessible resource (and vice versa).

On the one hand, the adversary can use control to direct the victim to an adversary-inaccessible resource when the program expects an adversary-accessible resource. The directory traversal and link following vulnerabilities of the classical *confused deputy* [16] (Row 2 in Table 1). On the other hand, the adversary can direct the victim to an adversary-accessible resource when the program expects an adversary-inaccessible resource. Trojan-horse libraries is an example vulnerability of this type. We call these *unexpected attack surface* vulnerabilities (Row 1 in Table 1), as they occur because the programmer is not expecting adversary control at these resource accesses. Table 1 classifies all resource access vulnerabilities into these two types.

2.2 Causes for Resource Access Vulnerabilities

We identify two causes for resource access vulnerabilities – one for each category in Table 1. The first cause is a mismatch in expectations of adversary control of names and bindings between the program and the deployment.

[2] Attacks involving names require the adversary to inject sequences of `../` or unicode characters. Thus, explicit data flow is necessary; just implicit data flow is insufficient.

Consider Figure 2 that describes resource accesses from the web server example in Figure 1. Here, the programmer expects the resource access of the HTML file to be under adversary control, and to combat this, adds a name filter [3] from the TCP socket (stripping `../`) as well as a binding filter (check for a link). The programmer did not expect the log file's resource access to be adversary-controlled, and therefore did not add any filters. However, due to a misconfiguration, this programmer expectation is not satisfied in the deployment configuration, causing a resource access vulnerability.

In general, resource access vulnerabilities are very challenging to eliminate because they involve multiple disconnected parties. First, programmers write code assuming that a certain subset of resource accesses are under adversarial control. Resource access checks cause overhead, so the programmer generally tries to minimize the number of checks, thereby motivating fewer filters. Second, there are OS distributors who define access control policies, thereby determining adversarial control of resource accesses. However, these OS distributors have little or no information about the assumptions the programmer has made about adversarial control, resulting in a set of possible mismatches. Finally, there are administrators who deploy the program on a concrete system. The configuration specifies the location of various resources such as log files. Thus, the administrator's configuration too may not match the programmer's expectation.

The second cause for resource access vulnerabilities is where the programmer does expect adversary-controlled resource access, but the filter may be insufficient to protect the program. Note that when a program encounters an adversary-controlled resource access, the only valid resource is an adversary-accessible resource; otherwise, the program is victim to a confused deputy vulnerability. Thus, the program needs to defend itself by filtering improper requests leading to a confused deputy. However, both name and binding filters are difficult to get right due to difficulty in string parsing [4] and inherent race conditions in the system call API [8] (e.g., lines 8, 9 in Figure 1).

In summary, the two causes of resource access vulnerabilities are: *(a)* unexpected adversarial control of resource access, and *(b)* improper filtering of resource access when adversary control of resource access is expected. These causes correspond to Rows 1 and 2 in Table 1 respectively. With these two causes in mind, we proceed to a model that precisely describes our solution.

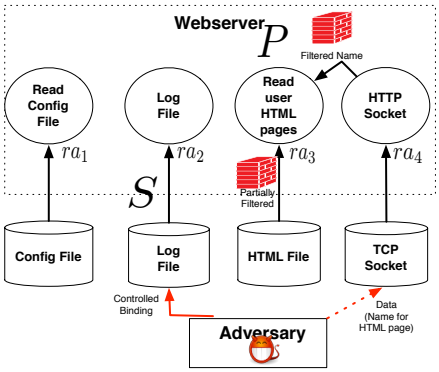


Figure 2: Demonstrating a mismatch between the expected and deployed attack surfaces.

3 Model and Solution Overview

In this section, we provide two invariants that directly address the two causes for resource access vulnerabilities outlined above.

Consider the set of all resource accesses *RA* made by a program[4]. A resource access happens when a system call resolves a resource using a name and namespace bindings. The program has code to filter the names and bindings used during certain resource accesses (e.g., ra_3 in Figure 2). From this knowledge, we show in Section 5 how to derive P, the set of resource accesses that a programmer expects to be under adversarial control. This set P is the *expected resource access attack surface*, or simply, the expected attack surface.

Now, assume that the program is deployed and run on a system. A subset of the resource accesses made by the program is adversary-controlled in the deployment. Let Y be the deployment's access control policy. Let S be the set of resource accesses that are adversary-controlled under Y (ra_2 in Figure 2). This set S defines the *deployment resource access attack surface*, or simply, the deployment attack surface.

Given Y, the expected attack surface P is *safe* for the deployment S if $S \subseteq P$, i.e., if all resource accesses in the deployment attack surface are part of the program's expected attack surface. Intuitively, this means that the program has filters to protect itself whenever a resource access is adversary-controlled. If r is the resource access under consideration, then, the invariant stated in propositional logic blocking unexpected adversary is:

$$\textbf{Invariant: Unexpected Adversary Control}(r): (r \in S) \rightarrow (r \in P) \quad (1)$$

If this safety invariant is enforced, resource access vul-

[3] A filter is a check in code that allows only a subset of names, bindings and resources through.

[4] The representation used to identify a resource access is implementation-dependent. In our implementation, we use program stacks at the time of the resource access system call.

nerabilities are eliminated where programs do not expect adversary control. Therefore, vulnerabilities are only possible where programs expect adversary control.

Now that we are dealing with an adversary-controlled resource access ($\in S$) that is also expected ($\in P$), the only valid resource is an adversary-accessible resource; otherwise, the program would be victim to a confused deputy attack. We say that resource accesses in P are protected from a confused deputy vulnerability if, when the resource access is adversary-controlled (i.e., $\in S$), it does not accept adversary-inaccessible resources. Let R be the set of resource accesses that retrieve adversary-accessible resources (as defined under Y). Then, a resource access r is protected from confused deputy vulnerabilities if the following invariant stated in proposition logic holds:

$$\textbf{Invariant: Confused Deputy}(r): \quad (r \in S) \rightarrow (r \in R) \tag{2}$$

Once these two rules are enforced, the only resources that are allowed are adversary-accessible resources where programs expect adversary control. Problems occur if the program does not properly handle this adversary-accessible resource. For example, if it does not filter data read from this resource properly, memory corruption vulnerabilities might result. Such vulnerabilities that occur in spite of fetching the expected resource are not within the scope of this work.

Let us examine how the rules above stop the vulnerability classes in Table 1. Consider vulnerability in Row 2, where the victim expects an adversary inaccessible resource (high integrity or secrecy), but ends up with an adversary-accessible (low integrity or secrecy) resource. The typical case is an untrusted search path where the program expects to load a high-integrity library, but searches for libraries in insecure paths due to programmer oversight or insecure environment variables, and ends up with a Trojan horse low-integrity library. Here, since the programmer does not expect a low-integrity library, she does not place a binding (or name) filter. Thus, we will infer that this resource access is not part of the expected attack surface ($\notin P$), and Invariant 1 above will stop the vulnerability if this resource access is controlled in any way (binding or name) by an adversary ($\in S$). The other vulnerability classes in this category are blocked similarly. Next, consider vulnerabilities in Row 1. Here, the victim expects an adversary-accessible resource (low integrity or secrecy), but ends up with an adversary inaccessible resource (high integrity or secrecy). In a link following vulnerability, the adversary creates a symbolic link to a high-secrecy or high-integrity file, such as the password file. Thus, the adversary uses her control of bindings ($\in S$) to direct the victim to an adversary-inaccessible resource ($\notin R$). In a directory traversal vulnerability, the adversary uses her control of the name to supply sequences of `../` to direct the victim to a high-secrecy or high-integrity file. In both cases, Invariant 2 will block the vulnerability since the adversary controls the resource access ($\in S$) through the name or binding, but the resource is adversary inaccessible ($\notin R$).

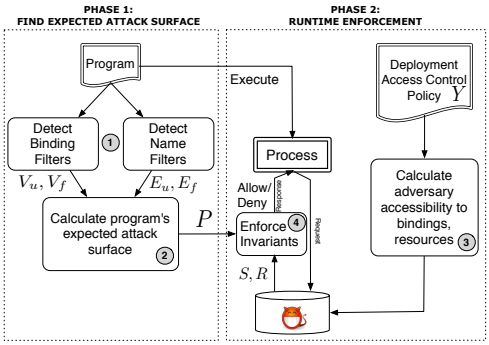


Figure 3: Overview of the design of our system.

4 JIGSAW Approach Overview

Figure 3 shows an outline of the design of JIGSAW. JIGSAW has two phases. In the first phase, JIGSAW calculates P, the expected attack surface. Finding P requires inferring programmer expectations. To infer programmer expectations, we propose an intuitive heuristic – if the programmer expects adversary control at a resource access, she will place filters in code to handle such control. Given the program, we perform a black-box analysis to detect the existence of any binding and name filtering separately (Step 1 in Figure 3), and use this information to calculate the program's expected attack surface (Step 2).

In the second phase, JIGSAW enforces Invariants 1 and 2 above by determining S, the deployment attack surface, and R, the set of adversary-accessible resources. The deployment's access control policy Y determines which resources and bindings are adversary-accessible. We leverage existing techniques to calculate adversary accessibility given Y [10, 17, 41] (Step 3). At runtime, if an adversary-accessible resource is used, that resource access is in R. If the name is read from an adversary-accessible resource or the binding used in resolving that name is adversary accessible, then that resource access is in S. Finally, we need to enforce Invariants 1 and 2 for individual resource accesses (Step 4). Any enforcement mechanism that applies distinct access control rules per individual resource access system call would be suitable. In our prototype implementation we leverage the open-source Process Firewall [40] which enables us to support binary-only programs (i.e., our prototype implementation does not rely on source code access).

5 Phase 1: Find Expected Attack Surfaces

The first step is to determine the expected attack surface P for a program. We do this in two parts. First, we propose a heuristic that implies the expectations of programmers with respect to the adversary control of the inputs to resource access and introduce the abstraction of a *name flow graph* to model these expectations and enable the detection of missing filters (Sections 5.1 to 5.3). Next, we outline how one can use dynamic analysis methods to build name flow graphs by accounting for adversary control of names and bindings (Sections 5.4 and 5.5).

5.1 Resource Access Expectations

Determining P requires knowledge of the programmers' expectations – whether the programmers expected the resource access to be under adversary control or not. The most precise solution to this problem is to ask each programmer to specify her expectation. Unfortunately, such annotations do not exist currently. As an alternative, we use the presence of code filters to infer programmer expectation. We use the following heuristic:

Heuristic. *If a programmer expects adversarial control of a resource access, she will add code filters to protect the program from adversarial control.*

Thus, the way we infer if a programmer expects an adversary-controlled resource access is by detecting if she adds *any* code to filter such adversarial control. An adversary controls a resource access by controlling either the name or a binding used in the resource access. Thus, we need to detect whether a program filters names and bindings separately.

Before presenting exactly how we detect filtering, we will introduce the concept of a *name flow graph* for a program, which we will use to derive the expected attack surface P given knowledge of filtered resource accesses.

5.2 Name Flow Graph

We introduce the abstraction of a name flow graph, which represents the data flow of name values among resource accesses in the program annotated with the knowledge of whether names and/or bindings are filtered each individual resource access. Using this name flow graph, we will show that we can compute resources accesses that are missing filters automatically. A name flow graph $G_n = (V, E)$ is a graph where the resource accesses are nodes and each edge $(a, b) \in E$ represents whether *there exists* a data flow in the program between the data of any of the resources retrieved at the access at node a and any of the name variables used in an access at node b. We refer to these edges as *name flows*.

Further, $V = V_f \cup V_u$ where V_f is the set of resource accesses that filter bindings and V_u the set of vertices that do not. Similarly, $E = E_f \cup E_u$, where E_f is the set of name flows that are filtered, and E_u the set that is not. That is, a name flow graph is a data-flow graph that captures the flow of names and is annotated with information about filters. The meaning of filtering for names and bindings is described in Sections 5.4 and 5.5, respectively.

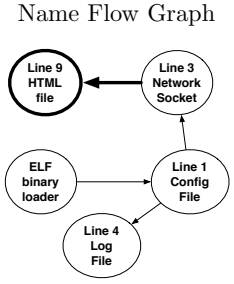


Figure 4: Name flow graph for the example in Figure 1.

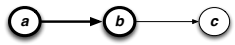
The name flow graph for our web server in Figure 1 is shown in Figure 4. Its nodes are resource accesses and edges connect two resource accesses if the data read at the source resource access may affect the name used at the target resource access. The bold nodes are those that filter bindings, whereas the bold edges are those that filter names.

The name flow graph determines P, the expected attack surface. According to our heuristic in Section 5, a resource access is part of the expected attack surface if a programmer places both name and binding filters on the resource access to handle adversarial control. However, not all name flows need be filtered – only name flows originating from other resource accesses also under adversarial control must be. Since this definition is transitive, we need to start with some initial information about resource accesses that are part of the expected attack surface, which we do not have. However, we find that we can easily define which resource accesses should *not* be in P. That is, we can use the *absence* of filters to determine resource accesses that should not be under adversarial control. This complement set of P is $\overline{P}$. We define an approach to calculate $\overline{P}$ below. Any resource access not in $\overline{P}$ is then in P, the expected attack surface.

Formally, a resource access $u \in \overline{P}$ if any of the following conditions are satisfied:

(i) $u \in V_u$: Binding filters do not exist, or
(ii) $u \xrightarrow{e} v \in V_u$: There exists an unfiltered name flow edge originating at u, or
(iii) $(u \xrightarrow{*} v) \wedge (v \in \overline{P})$: There exists a name flow path originating at u to a resource access in $\overline{P}$.

Consider the example in Figure 5. Here, resource accesses a and b filter bindings ($a, b \in V_f$). c does not filter



$a, b, c \in \overline{P}$

Figure 5: Example about determining membership in P

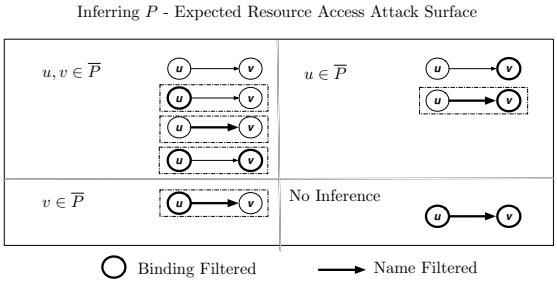


Figure 6: Determining whether a resource access in a resource flow graph should be in $\overline{P}$.

bindings ($c \in V_u$). c's name is determined from input at b, and b's name is determined from input at a. The name flow from a to b is filtered. By *(i)* above, $c \in \overline{P}$ since it does not filter bindings, and the programmer did not expect adversary control by our heuristic. Next, by *(ii)* above, $b \in \overline{P}$, since it is the origin of an unfiltered name flow (which adversaries should not control). Finally, by transitivity using *(iii)* above, $a \in \overline{P}$, because it is the origin of a name flow to a resource access that is in $\overline{P}$, and thus adversaries should not control the name read from resource access at a. All combinations of name and binding filters between a pair of nodes and the inference of node membership in P are presented in Figure 6.

Figure 7 describes the algorithm used to calculate membership in P, given V_f, V_u, E_f, and E_u. It implements *(i)-(iii)* above. It starts by initially assigning any node that does not filter bindings to $\overline{P}$ (*(i)*), and the source of unfiltered name flows to $\overline{P}$ (*(ii)*). It then uses a fixed point iteration to apply the transitive rule *(iii)*, and adds the source of any name flow to a target already in $\overline{P}$ to $\overline{P}$. At the termination of the algorithm, any resource access not in $\overline{P}$ is in P.

5.3 Detecting Missing Filters

Using the name flow graph, we can compute cases where filtering is likely missing. Intuitively, a filter is missing if the program filters *some* adversarial control of resource access but not others. This can happen in two cases: *(a)* if an incoming name flow is filtered but the binding at the resource access is not, or *(b)* a binding is filtered but an outgoing name flow is not. The dotted boxes in Figure 6 show these cases.

Precisely, filters are possibly missing at a resource access r in two cases:

Case 1: $\exists s, e : (s \xrightarrow{e} r \wedge e \in E_f \wedge r \in V_u)$. There exists a filter on an incoming name flow (indicating adversarial control of name) but not a binding filter, or

Input: Set of unfiltered names E_u and bindings V_u
Output: P

```
1:  P ← ∅                          ▷ Resource accesses that can be adversary controlled
2:  for v ∈ V_u do                 ▷ Any node that does not filter bindings
3:      P̄ ← P̄ ∪ v                  ▷ Cannot be adversary controlled
4:  end for
5:  for e ∈ E_u do                 ▷ Any edge that does not filter name
6:      P̄ ← P̄ ∪ e.src              ▷ Mark source as not adversary controlled
7:  end for
8:  c ← True
9:  while c = True do              ▷ Propagate set - fixed point iteration
10:     c ← False
11:     for e ∈ E_u do
12:         if e.tgt ∈ P̄ ∧ e.src ∉ P̄ then
13:             P̄ ← P̄ ∪ e.src
14:             c ← True
15:         end if
16:     end for
17: end while
```

Figure 7: Inferring P from knowledge of filtering

Case 2: $\exists s, e : (r \xrightarrow{e} s \wedge e \in E_u \wedge r \in V_f)$. There exists a filter on a binding (indicating an adversary-accessible resource) but not on all outgoing name flows.

As an example of a missing filter indicating a vulnerability, we found that in the default configuration, the Apache web server filters the name supplied by a client (by stripping `../`), but does not filter the binding used to fetch the HTML file. Therefore, an adversary can create a link of her web page to `/etc/passwd`, which will be served.

Not all possibly missing filters indicate a vulnerability. Some filters perform the same checks as JIGSAW. As an example, we found that `libc` had binding filters when it accessed (some) resources under `/etc` to reject adversary-accessible resources, enforcing Invariant 1 itself. Thus, there is no need to filter names originating from this resource (although *Case 2* indicates a possibly missing filter). We call filters that perform the same checks JIGSAW, redundant.

5.4 Detecting Presence of Binding Filters

We now outline our technique for detecting the filtering of bindings. Our objective in detecting here is to determine resource accesses that perform *any* filtering of bindings. Note that we do *not* aim to prove the correctness of the filtering checks themselves.

To define how we detect binding filters, we discuss how bindings are involved in resource access and how programs filter them. A program accesses many bindings (directories and symbolic links) during a single resource access. In theory, any one of these is controllable by the adversary. Filtering of directories is done by checking its security label, whereas link filtering checks if the binding is a link, and optionally, the security label of the link's

target. Bindings are filtered if, in some cases, the program does not accept a binding based on checks done on any binding used during resource access. An ideal solution would detect the existence of any such check.

Both static and dynamic approaches are possible to detect binding filtering. Static analysis uses the program code to determine if checks exist. However, this is quite challenging as there are a wide variety of ways to perform checks, including, for example, lowering the privilege of the process to that of the adversary [9, 39]. Instead, we opt for a dynamic analysis that detects the effects of filtering.

To detect filters, we have to choose a test that will *definitely fire the filter*, if such a filter is present. Our tests are attacks that attempt to exploit vulnerabilities if filters were absent. Not all attacks corresponding to vulnerability classes in Table 1 are suitable as tests to detect program filters. Consider the subset of attacks corresponding to vulnerability classes in Table 1 where the adversary uses her control of bindings to direct the victim to an adversary-accessible resource (Row 1). If the program accepts the adversary-accessible resource, it is generally not possible to determine if this was due to the program intentionally accepting this resource or due to the program assuming that there would be no adversary control of the resource access. On the other hand, consider the subset of attacks corresponding to vulnerabilities where the adversary uses control of bindings to direct the victim to an adversary-inaccessible resource (e.g., link following). Here, if the programmer were expecting adversary-controlled bindings, she *has* to add checks to block this resource access as this scenario is, by definition, a confused deputy vulnerability. Thus, we can use the results of a link following vulnerability to determine the existence of binding filters, and thus, the programmer's expectation. In Section 8, we describe a dynamic analysis framework that performs these tests.

5.5 Detecting Presence of Name Filtering

The other way for adversaries to control resource access is to control names. We aim to determine if the program makes any attempt to filter names, which would indicate that the programmer expected to receive an adversary-controlled name. Again, note that to determine programmer expectation, we only need to determine if there is *any* filtering at all, not if the filtering is correct.

To determine name filters in programs, we first describe how names originate. Names are either hard-coded in the program or received at runtime. First, hard-coded names are constants defined in the program binary or a dynamic library. For an adversary to have control of hard-coded names, she needs to control the binary or library, in which case trivial code attacks are possible. Therefore, we assume hard-coded names to not be under adversarial control. Second, programs get names from runtime input. In Figure 1, a client is requesting a HTML file by supplying its name. The server reads the name from this request (name source) and accesses the HTML file resource from this client input (name sink). In general, a name can be computed from input using one or more read system calls.

Next, we define the action of filtering names. There are two ways in which programs filter names. First, programs can directly manipulate the name. For example, web servers strip malicious characters (e.g., `..`) from names. Second, it can check that the resource retrieved from this name is indeed accessible to the adversary. For example, the setuid `mount` program accepts a directory to mount from untrusted users who are potential adversaries, but checks that the user indeed has write permissions to the directory before mounting. Thus, a name is filtered between a source and a sink if, in some cases, the name read at a source is changed, or the resource access at the sink is blocked. An ideal solution would detect the existence of any such checks.

Determining name filtering is a two-step process. First, we should determine pairs of resource accesses where the name is read from one resource (source) and used in the other (sink). Next, we determine if the program places any filters between this source-sink pair.

Again, we can use static or dynamic analysis to find pairs and filters. To detect filters, Balzarotti et al. used string analysis [4], whereas techniques such as weakest preconditions [7] or symbolic execution [18] can also be used. However, static analysis techniques are traditionally sound, but may produce false positives. Therefore, we use dynamic analysis to detect evidence of filtering.

To determine both pairs and filtering, we use a runtime analysis inspired by Sekar [33]. Sekar's aim is to detect injection attacks in a black-box manner. The technique is to log all reads and writes by programs, and find a correlation between reads and writes using an approximate string matching algorithm. Thus, given as input a log of the program's read and write buffers, the algorithm returns true if a write buffer matches a read buffer "approximately".

We adapt this technique to find name flows. We log all reads and names during resource access, and find matches between names and read buffers. We first try matching the full name; if no match is found, we try to match the directory path and final resource separately. Often, parts of a name are read from data of different resources. For example, a web server's document root is read from the configuration file, whereas the file to be served is read from the client's input. Both of these are combined to form the name of the resource served. As with the method for finding binding filters, we use the directory traversal attack in Row 2 to trigger filtering.

Since our analysis is a black-box approach, if a possible name flow is found, the read buffer might just coincidentally happen to have the name, but not actually flow to it. Thus, we execute a verification step. We run the test suite again, but this time change the read buffer containing the name to special characters, noting if the name also changes. If it does, we have found a name flow.

6 Phase 2: Enforce Programmer Expectations

Once we find the expected attack surface P, JIGSAW enforces resource access protections using Invariant 1 and Invariant 2 in Section 3 on program deployments. To do this, a reference monitor [1] has to mediate all resource accesses and enforce these rules. To enforce these rules correctly for each resource access, a reference monitor must determine whether this resource access is in P, and identify the system deployment's attack surface S and adversary accessibility to resources R.

6.1 Protecting Accesses in P at Runtime

The first challenge is to determine whether the resource access is in P. There are two ways to do this: (a) the program code can be modified to convey its expectation to the monitor through APIs, or (b) the monitor already knows the program expectation and identifies each resource access. Capability systems use code to convey their expectation during each resource access. Capability systems [21] present capabilities for only the expected resources to the OS during access. For example, decentralized information flow control (DIFC) systems [19,45] require the programmer to choose labels for the authorized resource for each resource access. However, such systems require modifying program code and recompilation, which can be complex to do correctly.

Another option is for the reference monitor to extract information necessary for it to identify the specific resource access, and hence whether it is in P. Researchers have recently made the observation that if they only protect a process, they may introspect into the process (safely) to make protection decisions [40]. They implemented a mechanism called the Process Firewall, a Linux kernel module that introspects into the process to enforce rules to block vulnerabilities per system call invocation. This is similar in concept to a network firewall that protects a host by restricting access per individual firewall rules. We use this option because it does not require program code or system access control policy changes, and was shown to be much faster than corresponding program checks in some cases.

The general invariant that the Process Firewall enforces is as follows:

`pf_invariant(subject,` **entrypoint**`,` **syscall_trace**`,`
`object,` **resource_id**`,` **adversary_access**`, op)` $\mapsto$ `Y|N`

Here, **entrypoint** is the user stack at the time of the system call. Resource accesses in P are identified by their entrypoint. A single system call may access multiple bindings (e.g., directories and links) and a resource. As each binding and resource is accessed at runtime, its adversary access is used in the decision. If a binding is adversary-accessible, then the resource access is in S. If the final resource is adversary-accessible, then the resource access is in R. If a resource access in R is the source of a name, this fact is recorded in **syscall_trace** and the resource access using this name is in S. This general invariant is instantiated to enforce our invariants in Section 3. The invariants are converted into Process Firewall rules using templates (Section 8).

6.2 Finding Adversary Accessibility R

R is the set of resource accesses at runtime that use adversary accessible resources, and is required to enforce Invariant 2 in Section 3. Calculating R requires knowing: *(a)* who an adversary is, and *(b)* whether the adversary has permissions to access resources. We address these questions in turn.

There have been several heuristics to determine who an adversary is. Gokyo [17] uses the system's mandatory access control policy to determine the set of SELinux labels that are trusted for the system – the rest are adversarial. Vijayakumar *et al.* [41] extend this approach to identify per-program adversaries. Chari et al. [10] and Pu et al. [43] use a model based on discretionary access control – a process running with a particular user ID (UID) has as its adversaries any other UID, except for the superuser (`root`). We can use any of these approaches to define an adversary.

Second, we need to determine whether an adversary has permissions to resources. As discussed in Section 2, an adversary-accessible resource is one that the system's access control policy Y allows an adversary permissions to (read for secrecy vulnerabilities, write for integrity vulnerabilities, and execute for both). This can be queried directly from the access control policy Y. Any resource access at runtime that uses adversary-accessible resources is in R.

Some resources become adversary-accessible through indirect means. For example, programs log adversarial requests to log files. Thus, adversaries affect data in log files even if the access control policy does not give adversaries direct write permissions to log files. Such exceptional resources are currently manually added to R.

6.3 Finding Deployment Attack Surface S

The deployment attack surface S is the set of resource accesses a process performs at runtime that are adversary-controlled. An adversary can control resource accesses

by controlling either the name or a binding (or both). An adversary controls a binding if she uses her write permissions in a directory to create a binding. An adversary controls a name if the adversary has write permission to the resource the name is fetched from.

To determine adversary control of names, we need to detect if there is a data flow from adversary-supplied data to a name used in a resource access. Data flow is most often determined by taint tracking [5, 20, 22, 31]. However, taint tracking techniques have overheads ranging from 2× to 50× [28]. Instead, JIGSAW approximates data flow using control flow (**entrypoint**s – process stacks at the time of reading names and using names). Pairs of process stacks during read and resource access system calls are initially associated by detecting explicit data flow between these calls (Section 5.5). During enforcement, if the Process Firewall sees the same stacks, we assume an explicit data flow between the read and resource access system calls, and the resource access is in S.

6.4 Finding Vulnerabilities

We can use the rules generated to also find vulnerabilities. Vulnerabilities are detected whenever a resource access is denied by our rules but is allowed by the program.

We use the same dynamic analysis from test suites that we use to detect the presence of filters in Sections 5.4 and 5.5 to also test the program for vulnerabilities in our particular deployment. Instead of enforcing the rules, we compare denials by Invariant 1 or Invariant 2 in Section 3 with whether the program allows the resource access. If the rule denies resource access whereas the program accepts the resource, we flag a vulnerability. Note that this process locates vulnerabilities in our specific deployment; there might be other vulnerabilities in other deployments that we miss. In any case, our rules, if enforced, will protect these program vulnerabilities in any deployment.

7 Proving Security of Resource Access

In this section, we first argue that if P, S and R are calculated perfectly, then JIGSAW eliminates resource access vulnerabilities. Our argument is oracle-based; that is, we can reason about the correctness of our approach assuming the correctness of certain oracles on which it depends. Our argument is contingent on the correctness of the three oracles that determine: *(i)* program expectation for P, *(ii)* adversary accessibility of resources for R, and *(iii)* adversary control of names and bindings for S. We then discuss the practical limitations JIGSAW faces in realizing these oracles.

7.1 Theoretical Argument

Assuming ideally correct oracles for determining programmer expectation, adversary accessibility of resources and adversary control of names and bindings, we argue that Invariants 1 and 2 in Section 3 protect a program from all resource access vulnerabilities as defined in Section 2.1 without false positives.

According to the definition in Section 2.1, a resource access vulnerability is caused when an adversary controls an input (name or binding) to direct a program to an adversary-accessible resource when the program expects an adversary-inaccessible resource (and vice-versa). Our proof hinges on two observations. First, resource access vulnerabilities are impossible if adversaries do not control the input name or binding. Invariant 1 denies all adversary control of inputs where the programmer expects only adversary-inaccessible resources, thus eliminating all vulnerabilities in these cases. Thus, vulnerabilities are only possible where the programmer expects to adversary control of input name or binding. Second, if indeed the adversary controls the input name or binding, the only authorized output is an adversary-accessible resource; otherwise, a confused deputy vulnerability (Row 2 in Table 1) can result. To block this, Invariant 2 allows retrieval of only adversary-accessible resources when input is under adversary control. Hence, we have shown that our rules deny resource accesses *if and only if* adversary control of input directs the program to unexpected resources, thus blocking resource access vulnerabilities without false positives.

7.2 Practical Limitations

In a practical setting, the determination of program expectations, adversary accessibility to resources, and adversary control of names and bindings is imperfect. This may lead to false positives and false negatives. We will discuss limitations with determining each of these in turn.

The first oracle determines programmer expectation, for which we use the intuitive heuristic in Section 5: if a programmer does not place filters, then she does not expect adversary control of resource access, i.e., she only expects adversary-inaccessible resources. The detection of filters themselves uses runtime analysis. This faces three issues: *(i)* if identified filters are actually present, *(ii)* if actual filters are missed, and *(iii)* incompleteness of runtime analysis. First, if a detected filter is not actually present, this might result in false negatives. However, to detect filters, we mimic an attack and detect if the program blocks the attack. The only way the program could have blocked the attack is if it had a filter. Second, if an actual filter is missed, this might result in

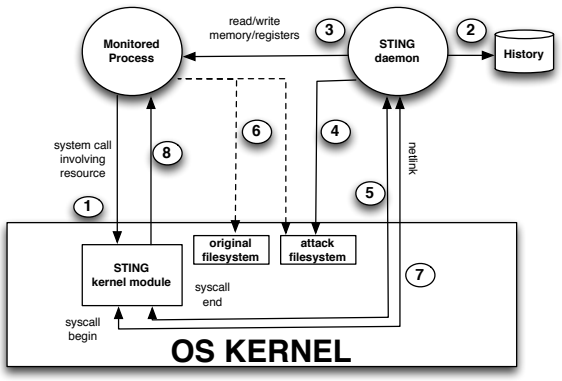


Figure 8: Implementation of JIGSAW's testing framework.

false positives. By the same argument above, if the filter cannot defend against our mimicked attack, then it is not complete enough anyway. Third, runtime analysis is inherently incomplete and may lead to false negatives for those resource accesses not covered. However, even with the limited developer test suites currently available, we were able to generate rules to block many vulnerabilities and find previously-unknown ones even in mature programs.

Next, we have the oracle that determines adversary accessibility to resources. The main challenge here is determining who an adversary is. If we do not have a sufficiently strong adversary model, we may miss adversaries and hence have false negatives. While there is no universally agreed-upon definition of who an adversary is, we use the intuitive DAC model[5] that most programmers assume [10, 43]. However, our framework permits the use of different adversary models. More conservative adversary models [41] will identify more adversary-accessible resources, possibly exposing more vulnerabilities.

The final oracle determines adversary control of names and bindings. The challenge here is to determine if there is a data flow from adversary-supplied data to a name used in a resource access. As described in Section 6.3, JIGSAW approximates data flow using control flow. However, even if control flow is the same across two executions of the program, it does not necessarily imply the data flow is the same, leading to false positives and negatives. While we have not found this to be a problem in our experiments (Section 9.4), more precise data flow tracking techniques [5, 20, 31] will address this challenge.

8 Implementation

There are two parts to our implementation. First, we need to test individual program resource accesses to detect the presence of filters. This is used by the algorithm in Figure 7 to generate P. Second, we need to enforce invariants in Section 3 using the Process Firewall. This involves determining R and S; this is done as discussed in Section 6.2 and Section 6.3 respectively.

8.1 Testing Programs

To test programs, we develop a framework that can intercept system calls and pass control to a user-space component that performs the tests. The kernel component is a small module that intercepts system calls and returns, and forwards them to a user-space daemon through netlink sockets. The flow of operations is as shown in Figure 8. When a monitored program makes a system call, it is intercepted by the framework's kernel module, and forwarded to the user-space daemon. There are two resource namespaces available per program – a "test" namespace that is modified for hypothetical tests and the original namespace (similar in principle to [39]). This daemon introspects into the monitored process to identify its resource access (using the user-space stack), and checks its history to see if filters have already been detected. If not, it then proceeds to modify the test filesystem (for binding filter detection). It then returns to the kernel module. Control passes to the process in kernel mode, which accesses the original or test filesystem (depending on whether binding filters are being tested). The system call end is also intercepted, and similarly forwarded to the user-space daemon to test for name filters (as the read buffer is now available and can be modified).

We use test suites provided with program source code to drive the programs. We repeatedly run these suites until all resource accesses have been tested for filters.

8.2 Enforcing Invariants

As noted, JIGSAW uses the open-source Process Firewall [40] to perform enforcement. The Process Firewall is a kernel module that uses the Linux Security Modules to mediate resource accesses. In addition, it can perform a user stack backtrace to identify the particular resource access being made. Given P and the edges in the name flow graph, we have two rule templates to instantiate invariants into rules to be enforced by the Process Firewall. Figure 9 shows the templates. Note that the rules for confused deputy are stateful. Adversary control of name or binding is recorded by the first rule, the adversary's identity is recorded by the second rule, and the third rule uses

[5] A process with uid X has as its adversaries any uid $Y \neq X$ (except superuser `root`)

Rule Templates

Unexpected Adversary Control: $(r \in S) \wedge (r \in \overline{P}) \implies Deny$
For each $r \in \overline{P}$:
```
pftables -i r.ept -d LOW -o DIR_SEARCH -j DROP
```

Confused Deputy: $(r \in P) \wedge (r \in S) \wedge (r \in \overline{R}) \implies Deny$
Name:
For each $r_1 \in P$ such that $E(r_1, r_2)$:
```
pftables -i r1.ept -d LOW -j STATE --set --key
<random_value> --value 1
pftables -i r1.ept -d LOW -j STATE --set --key "adv"
--value ADV_ID
pftables -i r2.ept -d HIGH -m STATE --key
<random_value> --cmp 1 --equal -m PERM -s --key
"adv" --deny -j DROP
```
Binding:
```
pftables -d LOW -o DIR_SEARCH -j STATE --set --key
<r1.ept> --value 1
pftables -d LOW -o DIR_SEARCH -j STATE --set --key
"adv" --value ADV_ID
pftables -d HIGH -m STATE --key <random_value> --cmp
1 --equal -m PERM -s --key "adv" --deny -j DROP
```

Figure 9: Process Firewall rule templates.

this state to block access to an adversary-inaccessible resource. Thus, R is adversary-specific; in addition to protecting programs against all its adversaries, it also prevents one adversary from using the program as a confused deputy against another adversary.

9 Evaluation

In this section, we evaluate our technique on several widely-used programs. We chose these programs because: *(i)* resource accesses are central to their operation, and *(ii)* they offer a study in contrast – OpenSSH and Postfix were explicitly architected for security [30], whereas the others were not. To derive expectations, we used developer test suites that came with the program or created our own. We answer: (a) how common are implicit programmer expectations during resource access, (b) whether the resulting expected attack surface was safe for our deployment and vulnerabilities where not, and (c) security effectiveness of hardened programs from resource access vulnerabilities. We find that in all programs except OpenSSH, more than 55% of all resource accesses are implicitly expected to be free from adversarial control. Moreover, we discovered two previously-unknown vulnerabilities and one default misconfiguration in the Apache webserver. Finally, we find that protection can be enforced with an overhead of <6% on a variety of programs and few false positives.

9.1 Implicit Programmer Expectations

Table 2 shows a summary of the results obtained by JIGSAW. We first note the percentage of resource accesses

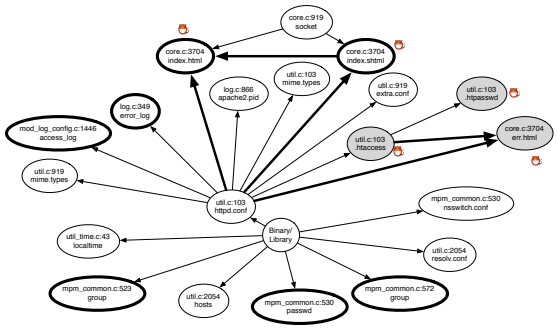


Figure 10: Resource Flow Graph for Apache. Nodes that have the icon of adversaries beside them are those where we found adversarial control of resource access in our deployment.

that are implicitly expected to *not* be in P, due to the absence of name or binding filters. For all programs except OpenSSH, the programmer placed no filters for more than 55% of resource accesses. If any of these resource accesses somehow come under adversarial control, the program can be compromised. It is very easy for OS distributor policies or administrator configurations to not match with these programmer assumptions. By explicitly identifying such resource accesses, we are able to protect them from any adversarial access in *any* deployment. OpenSSH makes fewer assumptions during resource access (17.6%). However, OpenSSH was re-architected after several years of experience with previous vulnerabilities. Using our technique, we can protect all resource accesses.

9.2 Case Study: Apache

In total, we found 20 resource accesses for Apache. Of these, Apache code filtered bindings for 7 accesses, and the name for 5 accesses. This led to 13 out of 20 resource accesses (65%) *not* being in P (using the algorithm in Figure 7). We found three resource accesses in $S-P$ for the Apache web server in our deployment, violating the first rule in Section 3. These corresponded to two previously unknown vulnerabilities in the Apache web server and one default misconfiguration. That such problems occur in even a mature program like Apache shows the importance of principled reasoning of resource access. While we found these vulnerabilities in our deployment, other deployments may have different vulnerabilities, but all will be blocked using our enforcement (Section 9.3).

Figure 10 shows the resource flow graph for Apache. Apache's expected attack surface is centered around resource accesses during the interaction with a client to serve a web page. It assumes that the location of the main configuration file and resources specified in it are not adversary controlled. Apache's resource flow graph is relatively complex due to long chains of resource flows, and it is difficult to reason about safety without the help of automated tools like ours. Resource accesses that had vul-

Program	*Dev Tests?*	$\|V\|$	$\|E\|$	$\|V_f\|$	$\|E_f\|$	$\in P$	$\notin P$	*Impl. Exp. %*	*Missing*	*Redundant*	*Vulns.*	*Inv. 1s*	*Inv. 2s*
Apache v2.2.22	Yes*	20	23	7	5	7	13	65%	2	0	3	13	12
OpenSSH v5.3p1	Yes	17	17	14	0	14	3	17.6%	0	3	0	3	2
Samba3 v3.4.7	Yes	210	84	78	19	78	132	62.8%	0	5	0	132	40
Winbind v3.4.7	Yes	50	38	19	13	19	31	63.3%	0	0	0	31	28
Postfix v2.10.0	No	181	15	79	7	79	102	56.32%	0	9	0	102	15

Table 2: Statistics of program-wide resource accesses. *Dev Tests* show whether we used developer test suites or created our own. *Impl. Exp.* is the percentage of resource accesses ($|\overline{P}|/|V|$) that are implicitly expected to be adversary-inaccessible. The last two columns show the number of instantiations of Invariant 1 and Invariant 2 in Section 3 for resource accesses in the program. *- We augmented the Apache test suite with additional tests.

nerabilities in our deployment are shaded in the graph.

The first vulnerability we found was during resource access of a user-defined `.htpasswd` file. Apache allows each user the option of enabling HTTP authentication for parts of their website. This includes the ability to specify a password file of their choice. However, the resource access that fetches this password file is not filtered. Thus, users can specify any password file – even one that they do not have access to. One example exploit is to direct this password file to be the system-wide `/etc/passwd`. Traditionally, it is difficult to brute-force the system-wide password file since prompts are rate-limited. However, since HTTP authentication is not rate-limited, this may make such brute-force attacks realistic. Such a scenario, though obvious after discovery, is very difficult to reason about manually due to Apache's complex resource accesses. Thus, it has remained hidden all these years.

The second vulnerability is a default misconfiguration. When serving web pages, Apache controls whether symbolic links can be followed from user web pages by the option `FollowSymLinks`, which is turned on by default in Ubuntu and Fedora packages. Turning this option on implicitly assumes trust in the user to link to only her own web pages. Interestingly, the name for this resource access is filtered – only the bindings are not. One way we were able to take advantage of this misconfiguration was through the error document on specific errors, such as HTTP 404, that is specifiable in the user-defined configuration `.htaccess` file. This allows an adversary to access any resource the Apache web server itself can read, for example, the password file and SSL private keys. We found that our department web server also had this option turned on. By simply making an error document linked to `/etc/passwd`, we were able to view the contents of the password file on the server. This demonstrates another typical cause of resource access attacks – administrators misconfiguring the program and violating safety of the expected attack surface.

The third vulnerability is a link following attack on `.htaccess`. Apache allows `.htaccess` to be any file on the filesystem it has access to. This may be exploited to leak configuration information about the webserver.

Finally, we note that test suites that come with programs are traditionally focussed towards testing functionality and not necessarily resource access. For example, the stock test suite for Apache only uncovered 7 resource accesses in total, and after we augmented it, there were 20 in total. Better test suites for resource access would help test more resource accesses.

9.3 Process Firewall Enforcement

Process Firewall rules enforce the safety of the expected attack surface under the deployment attack surface. Given the program's expected attack surface, Process Firewall rules enforce that any adversary-controlled resource access at runtime (i.e., part of the deployment attack surface) is allowed only if the resource access is also part of the program's expected attack surface. In addition, for those resource accesses allowed, they also protect the program against confused-deputy link and directory traversal vulnerabilities. The last two columns in Table 2 shows the number of Process Firewall rules we obtained (separately due to Invariants 1 and 2).

We evaluated the ability of rules to block vulnerabilities. First, we verified the ability of these rules to block the three discovered vulnerabilities in Apache. Second, we tried previously-known, representative resource access vulnerabilities against Apache and Samba. We tested an untrusted library load (CVE-2006-1564) against Apache. Here, a bug in the package manager forced Apache to search for modules in untrusted directories. Our tool deduced that the resource access that loaded libraries did not have any filtering, and thus, was not in P, blocking this vulnerability due to Invariant 1 in Section 3. In addition, we tested a directory traversal vulnerability in Samba (CVE-2010-0926). This is a confused deputy vulnerability involving a sequence of `../` in a symbolic link. This vulnerability was blocked due to Invariant 2.

9.4 False Positives

As discussed in Section 7.2, false positives are caused by improper determination of: *(i)* programmer expectation and *(ii)* adversary control of names.

First, false positives are caused by a failure of our heuristic in Section 3 that determines program expectation. In some cases, we found that a program had no filters at a resource access, but still expected adversary-controlled resource access. We found that this case occurs in certain small "helper" programs that perform a requested task on a resource without any resource access filters. For example, consider that the administrator (`root`) runs the `cat` utility on a file in an adversarial user's home directory. Because `cat` does not filter the input bindings, the user can always launch a link following attack by linking the file to the password file, for example. However, if there is no attempted attack, then our rule will block `cat` from accessing the user's file, because the resource access has no filters and is thus not part of the expected attack surface (by our heuristic). However, we may want to allow such access, because `cat` has filters to protect itself from the input data to prevent vulnerabilities such as buffer overflows.

To address such false positives, we propose enforcing protection for such helper programs specially. Our intuition is that when these programs perform adversary-controlled resource access, they can be considered adversarial themselves. All subsequent resources to which data is output by these programs are then considered adversary-accessible. Other programs reading these resources should protect themselves from input (e.g., names) as if they were dealing with an adversary-accessible resource.

To enforce this approach, we implemented two changes. First, we enforce only Invariant 2 (confused deputy) in Section 3 for these programs. Second, whenever Invariant 1 would have disallowed access, we instead allow access, but "taint" all subsequent output resources by marking them with the adversary's label (using filesystem extended attributes).

We evaluated the effect of this approach during the bootup sequence of our Ubuntu 10.04 LTS system. We manually identified 15 helper programs. During boot, various startup scripts invoked these helper programs a total of 36 times. In our deployment, 9 of these invocations accessed an adversary-accessible resource. Note that our original approach would have blocked these 9 resource accesses, disrupting the boot sequence, whereas our modification allows these resource accesses. These invocations subsequently tainted 4 output resources – two log files and two files storing runtime state. We found two programs reading from these tainted files – `ufw` (a simplified firewall), and the `wpa_supplicant` daemon (used to manage wireless connections). These programs will find the tainted resources adversary-accessible, and will have to protect themselves from such input.

Second, false positives are caused during enforcement by our implementation's approximation of adversarial data flow using control flow. Such false positives are due to implementation limitations and not any fundamental shortcoming of our approach. To evaluate this, we used two separate test suites for Apache – one to build the name flow graph and generate Process Firewall rules, and the other to test the produced rules. We used our enhanced test suite to generate rules and ApacheBench to test the generated rules. ApacheBench ran without any false positives. However, different configurations or variable values might result in different data flows even if the stacks remain the same. As mentioned in Section 7.2, accurate data flow tracking can solve this problem.

9.5 Performance

A detailed study of the Process Firewall is in [40]. In summary, system call microbenchmarks showed overheads of up to 10.6%, whereas macrobenchmarks had overheads of up to 4%. The main cause for overhead is unrolling the process stack to identify the system call. To confirm these results, we evaluated the performance overhead of a hardened Apache webserver (v2.2.22) that had the 25 rules from Table 2. Using ApacheBench to request the default Apache static webpage, we found an overhead of 4.33% and 5.28% for 1 and 100 concurrent clients respectively. However, we can compensate for such overhead by compiling Apache without resource access filters, since filters are now redundant given our enforced rules. Vijayakumar *et al.* [40] showed that removing code filters causes a throughput increase of up to 8% in Apache.

10 Related Work

10.1 Inferring Expectations

Determining programmer expectations from code has previously been done in a variety of contexts. Engler [14] et al. infer programmer *beliefs* from code. For example, if a pointer `p` is dereferenced, the inferred belief is that `p != NULL`. They use this to find bugs in the Linux kernel. Closely related are techniques to infer invariants from dynamic traces [3, 15, 32]. Daikon [15] uses dynamic traces to infer hypothesis such as that a particular variable is less than another. Baliga *et al.* [3] use Linux kernel traces to propose invariants on data structures. While we deal with a different class of attacks, our approach is in the same spirit as the above works.

10.2 Defenses During Resource Access

Current defenses against resource access attacks in OSes are ad-hoc and fundamentally limited. Defenses can be classified into those that require changes to either the *(i)* system (e.g., OS, libraries), or *(ii)* program code.

The simplest system defense is to change the access control policy to allow a process access to only the set of expected resources. Unfortunately, this defense is both complicated and does not entirely stop resource access attacks. First, fixing access control policies is a complicated task. For example, even the minimal (targeted) SELinux MAC policy on the widely-used Fedora Linux distribution has 95,600 rules. Understanding and changing such rules requires domain specific expertise that not all administrators have. Second, access control alone is insufficient to stop resource access attacks because it treats processes as a black-box and does not differentiate between different resource access system calls. In our example in Figure 1, the web server opens both a log file and user HTML pages. Thus, it needs permissions to both resources. However, it should not access the log file when it is serving a user HTML page, and vice versa. Traditional access control does not make this difference. Other system defenses have mainly focused on TOCTTOU attacks [11,13,25–27,35–38,44] and link following [10]. However, system defenses are prone to cause false positives because they do not know what programs expect [8], e.g., which pairs of system calls expect to access the same resource.

The simplest program defense is to use program code filters that accept only the expected resources. However, there are a number of challenges to writing correct code checks. First, such checks are *inefficient* and cause performance overhead. For example, the Apache web server documentation [2] recommends switching off resource access checks during web page file retrieval to improve performance. Second, checks are *complicated*. The system-call API that programs use for resource access is not atomic, leading to TOCTTOU races. There is no known race-free method to perform an `access-open` check in the current system call API [8]. Chari *et al.* [10] show that to defend link following attacks, programmers must perform at least four additional system calls per path component for each resource access. Going back to the example in Figure 1, the checks on lines 7 and 8 are not enough – the correct sequence to use is `lstat-open-fstat-lstat` [10]. Thirdly, program checks are *incomplete*, because adversary accessibility to resources is not sufficiently exposed to programs by the system-call API. Currently, programs can query adversary accessibility only for UNIX discretionary access control (DAC) policies (e.g., using the `access` system call), but many UNIX systems now also enforce mandatory access control (MAC) policies (e.g., SELinux [24] and AppArmor [23]) that allow different adversary accessibility. While custom APIs have been proposed [19,29,34,42] to address such limitations, these require additional programmer effort and do not protect current programs.

11 Conclusion

In this paper, we presented JIGSAW, an automated approach to protect programs from resource access attacks. We first precisely defined resource access attacks, and then noted the fundamental cause for them – a mismatch between programmer expectations and the actual deployment the program runs in. We defined two invariants that, if evaluated and enforced correctly, can theoretically offer complete protection against resource access attacks. We proposed a novel technique to evaluate programmer expectations based on the presence of filters, and showed how the invariants could practically be enforced by the Process Firewall.

We applied this technique to harden widely-used programs, and discovered that programmers make a lot of implicit assumptions. In this process, we discovered two previously-unknown exploitable vulnerabilities as well as a default misconfiguration in Apache, the world's most widely used web server. This shows that even mature programs only reason about resource access in an ad-hoc manner. The analysis as presented in this paper can thus efficiently and automatically protect programs against resource attacks at runtime.

Acknowledgements

We thank the anonymous reviewers and our shepherd David Evans for their insightful comments that helped improve the presentation of the paper. Authors from Penn State acknowledge support from the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-FA9550-12-1-0166. Mathias Payer was supported through the DARPA award HR0011-12-2-005. 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.

References

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

[2] Apache Performance Tuning. `http://httpd.apache.org/docs/2.2/misc/perf-tuning.html#symlinks`, 2012.

[3] A. Baliga, V. Ganapathy, and L. Iftode. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, pages 77–86, Anaheim, California, USA, December 2008. IEEE Computer Society Press, Los Alamitos, California, USA.

[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] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A system to specify and manage multipolicy access control models. In *Proceedings of POLICY'02*. IEEE Computer Society, 2002.

[6] M. Bishop and M. Digler. Checking for race conditions in file accesses. *Computer Systems*, 9(2), Spring 1996.

[7] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, 2007.

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

[9] S. Chari and P. Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Transaction on Infomation and System Security*, 6:173–200, May 2003.

[10] S. Chari *et al.* Where do you want to go today? escalating privileges by pathname manipulation. In *NDSS '10*, 2010.

[11] C. Cowan, S. Beattie, C. Wright, and G. Kroah-Hartman. Raceguard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[12] Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[13] D. Dean and A. Hu. Fixing races for fun and profit. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[16] N. Hardy. The confused deputy. *Operating Systems Review*, 22(4):36–38, Oct. 1988.

[17] T. Jaeger, R. Sailer, and X. 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.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

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

[20] L. C. Lam and T.-c. Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of ACSAC '06*, pages 463–472. IEEE Computer Society, 2006.

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

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

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

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

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

[26] 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.

[27] M. Payer and T. R. Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, 2012.

[28] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 115–126, Berkeley, CA, USA, 2013. USENIX Association.

[29] D. E. Porter *et al.* Operating system transactions. In *SOSP '09*, 2009.

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

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

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[33] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[34] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, December 1999.

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

[36] D. Tsafrir *et al.* Portably solving file tocttou races with hardness amplification. In *USENIX FAST*, 2008.

[37] 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.

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

[39] H. Vijayakumar, J. Schiffman, and T. Jaeger. Sting: Finding name resolution vulnerabilities in programs. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 2012)*, August 2012.

[40] 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 (EUROSYS 2013)*, April 2013.

[41] H. Vijayakumar *et al.* Integrity walls: Finding attack surfaces from mandatory access control policies. In *ASIACCS*, 2012.

[42] R. Watson *et al.* Capsicum: practical capabilities for UNIX. In *USENIX Security*, 2010.

[43] J. Wei *et al.* Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *USENIX FAST '05*, 2005.

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

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