The Right Files at the Right Time

Hayawardh Vijayakumar and Trent Jaeger
Systems and Internet Infrastructure Security Lab,
The Pennsylvania State University,
University Park, PA 16802
Email: {hvijay, tjaeger}@cse.psu.edu

***Abstract*—Programs fetch resources, such as files, from the operating system through the process of *name resolution*. However, name resolution can be subverted by adversaries to redirect victim processes to resources chosen by the adversaries, leading to a variety of attacks. These attacks are possible because traditional access control treats processes as black boxes, permitting all process permissions to all process system calls, enabling adversaries to trick victims into using resources that are not appropriate for particular system calls. Researchers have examined methods for enforcing distinct policies on individual system calls, but these methods are difficult to use because programmers must specify which permissions apply when manually. In this work, we examine the generation of system call-specific program policies to augment access control to defend against such name resolution attacks. Our insight in this paper is that system calls can be classified by the properties of the resources accessed to produce policies automatically. Given specific knowledge about name resolution attacks, such a classification may be refined further to prevent many name resolution attacks with little chance of false positives. In this paper, we produce a policy using runtime analysis for an Ubuntu 12.04 distribution, finding that 98.5% of accesses can be restricted to prevent typical name resolution attacks and more than 65% of accesses can be restricted to a single file without creating false positives. We also examine three programs in detail to evaluate the efficacy of using the provided package test suites to generate policies, finding that administrators can produce effective policies automatically.**

I. Introduction

Many vulnerabilities are caused because processes are tricked into using the wrong file for a particular task. In some cases, processes use adversary-controlled files when they expect protected files. For example, an untrusted search path vulnerability directs a process to an adversary-controlled file instead an expected library. In other cases, processes access trusted files when they expect unprivileged resources. Adversaries may redirect vulnerable processes to system files using links or maliciously-crafted file names using link and directory traversal attacks, respectively. We refer to these vulnerabilities collectively as *name resolution vulnerabilities*.

Authorization systems do not block access to name resolution vulnerabilities because they treat processes as black boxes. An authorization system restricts each process to perform only authorized operations on authorized objects given the process's subject, but any process system call can use any of these operations at any time. Processes are not homogenous entities, however. Each system call may have distinct expectations regarding the properties of the files used, as described above. For example, a process may require access to the password file for one system call (e.g., to authenticate users), but access to the password may not be appropriate for other system calls that retrieve content to be returned to a remote user. In general, if an access control policy permits access to any file that is inappropriate for even one system call, then the process may be vulnerable to attack.

Such problems cannot be prevented by proposed system defenses. Sandboxing [1], [2], [3], [4] limits the permissions of a process, but cannot prevent one file from being accessible in one system call, but not another. A variety of system defenses have been proposed to prevent exploitation of race conditions in time-of-check-to-time-of-use (TOCTTOU) attacks [5], [6], [7], [8], [9], [10]. However, researchers have found that systems defenses are fundamentally limited or incur false positives because they lack an understanding of process expectations [11].

As a result, the task falls to programmers to ensure that the files they use satisfy their requirements. However, this is a difficult task for programmers to get right. The fundamental problem for programmers is that they often do not know when their adversaries may have access to the files they request or the directories they use to retrieve files from names. This depends on the configuration of the system upon which the program is run. System call APIs have been extended to check file properties or for the following of links in name resolution, but these conditions are not always unsafe and are only useful in preventing some of these vulnerabilities. As a result, vulnerabilities are often present even when programmers use these APIs [12]. Capability systems [13] enable programmers to specify different permissions on different system calls, but capability systems are not commonly used in practice where programmers need to specify permissions manually, probably for the reasons above.

As a result, researchers are confident that many name resolution vulnerabilities could be prevented by enforcing system call policies, but no single system policy always applies and programmers are incapable of manually specifying policies correctly. As a result, we need methods to generate system call policies automatically. Such policies must be consistent with process expectations to prevent attacks without creating false positives [11]. Runtime analysis is now commonly used to produce access control policies [14], [15], [16]. However, runtime analysis is inherently unsound, so test cases must test enough of the program's behavior to avoid false positives and testing must distinguish safe from unsafe cases to avoid false negatives. Static analysis can be sound, but as described above, safe name resolution depends on both the system and the program, making tractable static analysis difficult.

Our insight in this paper is that programs often expect particular system calls to retrieve objects with the same properties on each use. We find that many name resolution system calls always retrieve the same file, files with the same security label, or files that are trusted by the program. For such system calls, invariants can be enforced that limit the files that may be accessed based on such classification. With further knowledge about name resolution attacks, classification may be refined further to prevent more attacks. For example, a single system call may only access trusted files, but if we know that the system call is prone to time-of-check-to-time-of-use (TOCTTOU) attacks [17], [5], then the files that may be accessed may be limited further (e.g., to those in a prior system call). Finally, knowledge that the program is run from the same configuration inputs may be used to further refine policy. For example, the same program run from two different configuration files may make distinct accesses as guided by the configuration file.

In this paper, we describe a method for generating system call policies automatically using runtime analysis. We produce system call policies for all programs the Ubuntu 12.04 distribution with a LAMP stack installed, finding that 98.5% of accesses can be restricted to prevent typical name resolution attacks and more than 65% of accesses can be restricted to a single file without creating false positives. As runtime analysis is unsound, we studied the use of package test suites for Apache, MySQL, and PHP. Results show how test suites can be helpful in covering more entrypoints and also exercising existing entrypoints in different ways. At the same time, we also find care must be taken to not allow test suites to generate false negatives for a particular deployment, and find that generating policies for a deployment can be automated.

II. Attacks

Once started, a process often needs additional system resources to complete any task (e.g., libraries, configuration files, logs, etc.) and may need to retrieve task-specific system resources (e.g., web content files, web requests via sockets, IPCs to worker processes, etc.). For convenience, resources are often retrieved by name, using a method known as *name resolution* [18], [19]. In a name resolution, a client (the process) provides a *name* to a name server (the OS), which uses *name bindings* managed by the name server to map the name to a *resource* managed by the OS. Various namespaces exist in operating systems, including the filesystem namespace, the signal namespace, and the System V IPC namespace in typical UNIX-based systems.

Name resolution attacks are possible because the names, name bindings, and resources used by the resolution mechanism may be controlled by adversaries. While programs may need to legitimately accept adversary interaction in certain name resolutions for functionality, problems arise when adversaries control name resolutions in ways the program does not expect. Table I shows examples of attacks that occur due to unexpected adversarial control of each of these components involved in name resolution.

Adversarial control of	**Attack Examples**
Name	Directory traversal (CWE-22), external control of filename or path (CWE-73)
Binding	Link following (CWE-59), TOCTTOU (CWE-367)
Resource	Resource squatting (CWE-283), untrusted search path (CWE-426), TOCTTOU (CWE-367)

TABLE I: Table showing example attacks (with CWE classes) that occur due to unexpected adversarial control of name resolution components.

```
01  read(client_socket, filename);
02  extn = extension(filename);
03  if (extn requires module m) {
04        load_module(m);
05        process(m, filename);
06  } else {
07        stat(filename, &buf);
08        if (filename not found) {
09                write(client_socket, "404 Not Found");
10        } else {
11                fd = open(filename, O_RDONLY);
12                write(client_socket, "200 OK");
13                write(client_socket, contents(fd));
14        }
15  }
```

Fig. 1: Simplified processing cycle of a typical webserver.

As an example illustrating these attacks, consider a simplified version of the processing loop of a typical webserver (e.g., Apache) in Figure 1. A remote client requests `filename` to be served. The webserver checks if a special module is required to serve the request (e.g., PHP for dynamic content), and loads it if required. Otherwise, it checks for the existence of the requested `filename`, and serves it if present. Possible adversaries of the webserver include both remote parties and local adversaries, such as dynamic content scripts supplied by untrusted parties. Only local adversaries are capable of modifying local information, such as name bindings and resources, whereas remote adversaries can modify names supplied to the server.

First, an adversary may attack the webserver by supplying malicious *names*. For example, `load_module` on line 4 searches for module files. Local adversaries have a variety of ways to affect the name used in such searches, using search path environment variables, insecure `RUNPATH` in binaries (CVE-2006-1564), and dynamic linker bugs. Remote adversaries may supply malicious names for the `filename`. For example, they may supply sequences of `../` to break out of the server's directory root and mount a directory traversal attack. If input filtering is not done properly, then the webserver may serve unauthorized files (line 14). While network filtering [20], [21] may block some attacks of this type, malicious names may not always be filtered correctly.

Handling names correctly has proven to be difficult for web application code (e.g., PHP inclusion attacks [22], [23]). Also, malicious names may obtained from local adversaries (e.g., untrusted configurations). Second, an adversary may attack the webserver by supplying malicious *name bindings*. Adversaries may supply symbolic links to redirect victims to sensitive resources, such as `/etc/shadow`. By default, Apache refuses to follow symbolic links in users' web directories to prevent this attack. This checking is done on line 7. However, a local adversary could change the name bindings corresponding to `filename` to a symbolic link to `/etc/shadow` between the check on line 7 and the use on line 11. Since the webserver is authorized to read `/etc/shadow`, this file will be opened on line 11, enabling leakage of secret data. This is an example of a classic time-of-check-to-time-of-use (TOCTTOU) attack [17], [5].

Third, an adversary may attack the webserver by controlling *resources* accessed by the webserver in unexpected ways. For example, if the webserver searches for modules in the user's document root directory, the user can supply a malicious library to gain control of the webserver process. Another example of unexpected control of a resource is IPC squatting, where the adversary creates a socket at a well-known location and masquerades as a legitimate server.

No current mechanism effectively prevents the myriad of name resolution attacks described above. Traditional access control is insufficient to prevent such attacks, as it views processes as a monolithic unit. In the example above, opening files in a user's web directory is valid on line 14, but invalid on line 4 while loading module libraries. Sandboxes [1], [2], [3], [4] have a similar limitation, as they may reduce process permissions, but they still view processes monolithically. Prior defenses against such attacks [8], [10], [24], [9], [25], [7], [6] have been found to be flawed or only cover a subset of these attacks under limited conditions [26], [11].

III. Related Work

Relevant prior work has shown us that having policies that provide distinct permissions for some system calls is valuable and that useful security policies can be produced in a mostly-automated way. However, current methods for producing policies are inadequate for preventing attacks on system calls without creating too many false positives and negatives.

a) System Call Enforcement: Research has identified the need to mediate process access to system resources independently per system call to prevent vulnerabilities. Sekar et al. [27] presented an intrusion detection system that used the instruction that invokes the system call library (which we call *entrypoint* below) to parameterize automata models of programs. However, such models are not directed towards attacks and do not scale. In addition, program-only models in general cannot prevent name resolution vulnerabilities, because programs may still use the same name.

More recently, methods to approximate classical integrity are capable of reasoning about individual program system calls [28], [29], [30], [31], [32], judging whether the program will be able to upgrade or discard inputs safely as required by Clark-Wilson integrity [33]. Some systems provide limited integrity protection by identifying objects that may affect the processes' integrity [28], [29]. However, if the process is allowed to use adversary-controlled objects and uses them at the wrong time, then vulnerabilities are likely. Some system calls may require protected objects only, but others may accept a variety of inputs. Other research requires programmers to describe the permissions per system call using annotations [30], [31], analogous to capabilities. Such policies are complex for programmers to specify correctly, as evidenced by name resolution vulnerabilities. Researchers have shown that such annotations can be produced from constraints [34], although programmers must still specify such constraints manually.

Researchers have also explored methods to enable programs to enforce access control policies [35], [36]. Such methods depend on programmers labeling the system call invocations through which untrusted inputs may be received. To improve the accuracy of such enforcement, researchers have also examined integrating system and program MAC enforcement [37], [38]. In this case, system and program policies must be integrated. In some cases, it is possible to automate such integration [39], although program policies are still uncommon.

b) Policy Generation: Historically, MAC policies, such as Bell-LaPadula [40], IX [41], and Caernarvon [42], must be specified manually. In modern commodity systems, however, runtime analysis is now commonly used to produce mandatory access control (MAC) policies [14], [16], [15]. Runtime analysis is primarily used to prevent false positives in policies, as early commodity MAC enforcement was shunned due to too many false positives. In these runtime analyses, security-critical programs are run, and the permissions that they use are logged. Any permission request logged is then granted for the program since some real program operation required the permission. As a result, these MAC enforcement policies are designed to satisfy *least privilege* [43], where processes are only granted the permissions that they require to perform their function. Since such policies are not produced from a security goal some permissions may be unsafe for the process. In addition, as discussed in the Section II, any process system call may use any of the permissions associated with the process, which causes the risks that adversaries leverage in name resolution attacks.

Runtime analysis is unsound, however, meaning that both false positives and false negatives are possible. False negatives may occur because even benign conditions may be unsafe. Suppose that the same system call is run in two different configurations where two different, but incompatible, files are accessed. Use of the wrong file may cause a vulnerability, which would be a false negative since both would be allowed. False positives may occur because of the lack of coverage in runtime test cases. If not enough cases are run, then the analysis may conclude that one file is accessed when in fact many may be accessed.

IV. Approach

To prevent name resolution attacks, we must limit the files that may be retrieved as part of a name resolution system call. Thus, a system call policy consists of a set of files that may be retrieved for a particular system call[1]. The challenge is to define system call subjects and determine a method to find the files that such subjects may access safely. As with commodity MAC policies, the goal is to minimize false positives.

Starting with system calls, we uniquely identify each system call invocation by its program entrypoint. A *program entrypoint* is the program instruction calls a function in the system call library that results in a system call invocation [44]. For example, when a program wants to open a new file, it invokes the `open` function in the system call library (e.g., libc). As the program may have many different instructions that request the `open` system call, the program entrypoint differentiates among them enabling distinct policies to be applied based on each.

Each program entrypoint may request one or more different files. We identify four distinct classifications: (1) same file[2]; (2) same label; (3) same integrity; and (4) any file. An entrypoint that is classified as *same file* is thought to retrieve one file in all cases. An entrypoint that is classified as *same label* is thought to retrieve multiple files, but each file has the same MAC label. An entrypoint that is classified as *same integrity* either only retrieves files that are assigned labels that are trusted by the program or untrusted by the program. Note that many name resolution attacks can be prevented simply by ensuring that system calls only retrieve trusted or untrusted files. The finer classifications provide further assurance that adversaries cannot redirect victim processes in an exploitable manner. The fourth classification is for entrypoints that may retrieve arbitrary files.

The classification of objects may be refined by knowledge about specific name resolution attacks, but we must be careful not to introduce false positives when using such knowledge. For example, TOCTTOU attacks change the file retrieved between a *check* system call and a corresponding *use* system call [24]. A check system call examines a file and the use system call enables the file to be processed. In general, a check-use pair is supposed to use the same file, so it is possible to restrict a use system call to a specific file (i.e., the one used in the check) even of the entrypoint was classified as any. Other attacks, such as PHP includes or directory traversal, may require limiting the file retrieved to a specific type and/or directory, although these restrictions are often represented in the MAC labeling of files.

Once the entrypoint is classified, then file access can be controlled per system call. While the approach above results in a low probably of false positives (depending on the analysis accuracy), some unnecessary false negatives may be created. The problem is that a single program may be run under multiple deployments. A program deployment may be influenced by a number of factors, such as its configuration files, environment variables, command line options, etc. The classifications may be refined further based on a specific program deployment, although the number of possible deployments for some programs may be large. Such deployment-specific policies are analogous to "booleans" in the SELinux policy, which specify different permissions when particular configuration options are selected. Note that booleans are set manually in SELinux policies, where we want find mappings between deployments and classifications automatically, if possible.

Finally, we use runtime analysis to produce the classification described above. As mentioned, runtime analysis may be error-prone because it is unsound, so it is important to find a method of producing runtime test cases with broad coverage of the functions performed by each program. Many program packages now include test suites used to test program functionality over a variety of configurations, so we explore the effectiveness of using such test suites for runtime analysis in this paper.

V. Design and Implementation

In this section, we discuss the design and implementation of our system to defend against name resolution attacks.

Our system is broadly divided into logging and enforcement phases. During the logging phase, our system logs accesses made by programs during name resolution calls, along with the entrypoint and security label of the final resource retrieved. Test suites for programs are run at this stage. These access logs are then used to classify entrypoints into four categories (Section VI), which are then enforced through rules that make sure that entrypoints obey their classification.

However, runtime coverage during the logging phase may not have captured all entrypoints possible, and test suites are not available for all programs. Thus, legitimate accesses may be blocked because they are not seen at runtime. This is a common problem amongst other runtime policy-generation approaches such as SELinux as well. Our approach is to allow any operation at previously unseen entrypoints; thus, the program will continue to function, although attacks may be missed.

Our system is implemented for the Linux 3.2 kernel. It layers on top of the SELinux access control module, which is itself called on Linux Security Module (LSM) hooks. Thus, we intercept security-sensitive operations, and our enforcement is done on top of access control.

Obtaining the entrypoint is done in the kernel by unwinding the userspace stack. However, straightforward unwinding fails in modern distributions as programs are compiled without frame pointers. To overcome this, we parse the `eh_frame` section of the ELF binary, which contains the necessary information to obtain the stack trace, and is compiled by default on modern Fedora and Ubuntu distributions.

If the program is an interpreter, obtaining the stack will only give the interpreter's entrypoint, and not the script file

[1]Name resolution attacks may be launched with any operation privilege to files, so we ignore the file operations requested in this work.

[2]This is actually the same inode, as inode is the unique identifier for file objects.

or line number. In previous work [44], we devised methods to introspect into Bash and PHP interpreters to obtain this information. Finally, the entrypoint for processes launched from the Bash shell interpreter contains the parent shell script's filename and line number. This is because the entrypoints for programs like `cp`, `mv` should be considered in relation to their parent script and not solely as separate programs.

VI. Evaluation

In this section, we first study the feasibility of using entrypoint classifications to restrict resource access. Our results suggest that most entrypoints can be classified as either accessing high or low integrity resources, leading to enforcement with few false positives. Next, test suites can help exercise programs to generate more accurate entrypoint resource mappings. We find that while test suites help significantly, they may also cause false negatives. Finally, we evaluate how effective our enforcement is in stopping attacks.

Our tests were carried out on an Ubuntu 12.04 Desktop distribution that also had the LAMP (Linux-Apache-MySQL-PHP) stack installed. The kernel had our module that layered on top of SELinux to perform logging and enforcement.

A. *Entrypoint classification*

Table II shows the classification of entrypoints exercised system-wide at normal runtime. Most entrypoints access very specific resources, and only a few (around 1.4%) of the total access resources of both high and low integrity. This was consistent even for programs run under the test suites – most entrypoints either accessed only high, or only low integrity. This suggests it is possible to constrain most entrypoints to either high or low resources, thereby enabling effective enforcement with low false positives.

Figure 2 shows the distribution of the number of resources accessed per entrypoint. More than 90% of entrypoints access 3 resources or less.

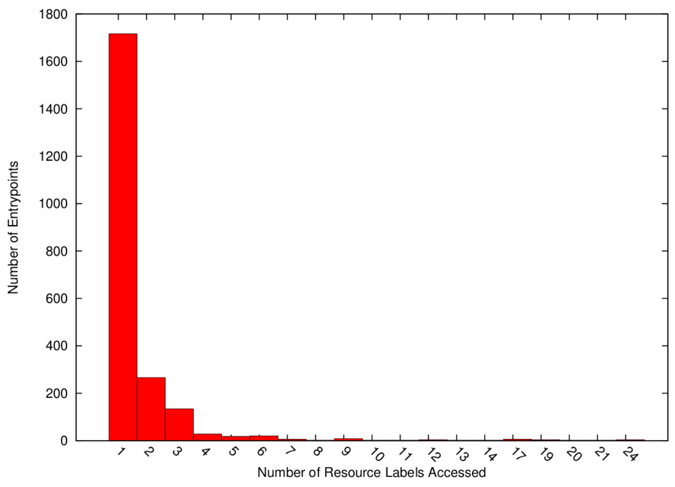

Fig. 2: Histogram showing the distribution of the number of resources accessed by entrypoints.

Entrypoint Class	Number	Percentage
Total	2196	-
Single filename	1486	67.6%
Single label	1716	78.1%
Only high-integrity	1910	86.9%
Only low-integrity	254	11.5%
Any integrity	32	1.5%

TABLE II: Different entrypoint classifications. It can be seen that only very few entrypoints access both high and low integrity resources.

Program	Normal Run	Test Suite	Class Change	
			FP Reduce	FN Increase
Apache	32	36	2	2
MySQL	12	14	0	0
PHP	33	48	0	1

TABLE III: Table showing the effect of test suites on programs. The first column shows the number of entrypoints exercised by a normal run, the second using the test suites. The last two columns show the number of entrypoints that changed classification after running the test suite, and whether they led to false positives, or negatives (identified manually)

B. *Test Suites*

We first examine how test suites can help generate accurate entrypoint to resource mappings by comparing normal runtime with test suites. Test suites contain test cases created by developers and are primarily geared towards testing functionality, although a few test suites also look for known security bugs. These test suites are usually meant to test binaries compiled from the package's source code, although many also support testing already deployed binaries. For example, test suites for the Apache webserver, PHP and MySQL support testing existing deployments, whereas the test suite for the OpenSSH server supports testing only a compiled binary. For normal runtime of the LAMP server stack, we installed phpBB, a PHP-based bulletin board system, and carried out tasks such as posting messages on the board. PHP was setup through FastCGI so it runs as a process separate from Apache.

Table III shows how test suites are helpful in identifying additional behaviour. In all cases, we found the test suites uncovered additional entrypoints beyond normal runtime, thereby leading to more complete code coverage. In addition, the test suites may exercise already known entrypoints in ways normal runtime does not, thereby augmenting the set of accessed resources at this entrypoint. Thus, while test suites reduce false positives by exercising known entrypoints in different ways, they may also introduce false negatives by accessing resources not valid in the actual deployment of the program.

C. *Factors Affecting Entrypoint Classification*

We found that the sets of resources accessed by entrypoints in programs are affected by the several factors. Test suites both uncover new entrypoints and exercise entrypoints in different ways by varying these factors. However, some of these factors are fixed for a deployment, and varying them may result in false negatives. We discuss each factor below in relation to Table III with examples.

Configuration. Some entrypoints only operate under certain configurations. This was one reason why additional entrypoints were uncovered by the test suites. However, such entrypoints may not be enabled at all in normal runtime.

Other entrypoints exercise different configurations. However, these might result in false negatives. As an example, the Apache configuration option `AllowOverride` allows Apache to accept user-defined configuration files for user web pages. This may cause a security threat if such configuration files are not handled properly. it was tested by the test suite, thereby classifying the entrypoint reading configurations as accessing both high-integrity system-defined configuration and low-integrity user-defined configuration. However, our deployed configuration did not allow this option. Here, the test suite caused a false negative by making the entrypoint classification more generic than it should be.

Rules should thus be generated keeping for a particular deployment configuration. However, knowledge of how configuration options affect entrypoints is beneficial. We propose "tagging" generated rules with the corresponding configuration options, so a rule base appropriate for a deployment can be generated by simply examining target configuration.

Command line parameters and Environment Variables. Varying command-line parameters can both reduce false positives and increase false negatives, similar to configuration options. This depends on whether a program is launched with differing arguments, or with a fixed set of command-line parameters (e.g., startup scripts).

First, changing command line parameters reduces false positives. For example, the `mount` program takes as a command line parameter a mountpoint. If runtime sees only a single invocation with a single mountpoint, it can erroneously conclude that the mount entrypoint can only access that particular mountpoint. However, during normal system boot, several mountpoints are used, and the mount entrypoint is classified as accessing both high- and low-integrity directories. Other utilities such as `cp, mv, cat` also exhibit similar behavior.

However, note that such programs are often launched from scripts where they have to access very particular files (e.g., redirect output to a temporary file). Thus, if a program is launched from a shell script, we take its entrypoint to be the parent interpreter's script, and can enforce such resource access.

Second, changing command line parameters increases false negatives. This behaviour is observed when command line parameters indicate configuration options that are not used. For example, the Apache test suite specifies a configuration file using a command line parameter, and runtime erroneously concludes that entrypoints accessing the log file are more generic than they should be.

Environment variables also have effects similar to command line parameters.

Working Directory. The working directory may affect the resources accessed if relative pathnames are specified. These again mainly affect utility programs.

Program	Reference	Class
Firefox	CVE-2010-3182	Untrusted Search
Apache	CVE-2006-1564	Untrusted Search
php	CVE-2006-5178	TOCTTOU
init script	Prev. unknown	Link following
php	CVE-2011-2202	Directory Traversal
mysql	CVE-2010-1848	Directory Traversal

TABLE IV: The exploits we tested our process firewall against.

D. *Effectiveness of Enforcement*

To evaluate the effectiveness of our enforcement on known bugs, we selected a few previously known vulnerabilities and a previously unknown and unpatched name resolution vulnerability [12] at entrypoints uncovered by our runtime analysis. Table IV shows the exact attacks we tried. We found that all the exploits that we tested were blocked.

Firefox and Apache searched for library files in the current working directory, which could lead to compromise if they were launched in an adversary-accessible directory. The entrypoint that reads library files was associated only with high-integrity files and so adversary supplied libraries at this entrypoint were blocked.

PHP is vulnerable to a directory traversal attack due to a parsing error in the filename of uploaded files (CVE-2011-2202). We tried to force a vulnerable version of PHP to store our uploaded file at an arbitrary location. This attack was also blocked because this entrypoint was associated only with resources in `/tmp`. This entrypoint was actually covered by the test suite and not normal runtime (as we did not upload any file during normal runtime), and is an example of how test suites help in covering program entrypoints. Finally, a shell script that initializes the avahi-daemon is vulnerable to link following due to insecurely writing to a temporary file. We created a symbolic link in `/tmp` to `/etc/passwd`. This attack was also blocked, because the script entrypoint was again associated with only `/tmp`, and access to `/etc` was disallowed. This demonstrates how our enforcement mechanism stops attacks, even though it aims for low false positives.

VII. Conclusions

The class of name resolution attacks is a difficult problem to solve, because it involves both program context and system knowledge. Thus, adversaries have been able to keep taking advantage of these vulnerabilities to compromise systems. We identify how adversarial control of the name, binding and resources used in name resolution can lead to a variety of attacks. We propose a uniform solution for these problems by restricting program entrypoints to only appropriate resources. We find that program entrypoints fall under classes which can be easily enforced to stop these attacks with little or no false positives. We find over 98% of name resolution accesses can be restricted to rule out name resolution attacks. Since our classification of entrypoints is based on a runtime analysis for which coverage is a challenge, we examine how test suites help to generate policies, and steps towards automation of policy generation. Finally, we demonstrate how our system can defend against instances of name resolution attacks, showing the promise of our system to protect programs against this class of attacks, while not unduly producing false positives. In future work, we aim to explore how static analysis can be used to generate rules.

References

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

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

[3] Acharya *et al.*, "MAPbox: Using parameterized behavior classes to confine untrusted applications," in *USENIX SSYM*, 2000.

[4] Garfinkel *et al.*, "Ostia: A delegating architecture for secure system call interposition," in *NDSS '04*, 2004.

[5] M. Bishop and M. Digler, "Checking for race conditions in file accesses," *Computer Systems*, vol. 9, no. 2, Spring 1996.

[6] C. Cowan *et al.*, "Raceguard: Kernel protection from temporary file race vulnerabilities," in *USENIX Security Symposium*, 2001.

[7] Tsyrklevich *et al*, "Dynamic detection and prevention of race conditions in file accesses," in *USENIX Security*, 2003.

[8] Dean *et al.*, "Fixing races for fun and profit," in *USENIX SSYM*, 2004.

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

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

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

[12] 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.

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

[14] N. Provos, "Improving host security with system call policies," in *USENIX Security '03*. USENIX Association, 2003.

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

[16] "audit2allow," http://fedoraproject.org/wiki/SELinux/audit2allow.

[17] W. S. McPhee, "Operating system integrity in OS/VS2," *IBM Syst. J.*, vol. 13, pp. 230–252, September 1974. [Online]. Available: http://dx.doi.org/10.1147/sj.133.0230

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

[19] "Domain Names - Implementation and Specification," http://www.ietf.org/rfc/rfc1035.txt.

[20] Vigna *et al.* , "Testing Network-based Intrusion Detection Signatures Using Mutant Exploits," in *ACM CCS*, 2004.

[21] "What is "Deep Inspection"?" http://http://www.ietf.org/rfc/rfc1035.txt.

[22] "PHP LFI to arbitrary code execution. ," http://www.exploit-db.com/download_pdf/17010/.

[23] Balzarotti *et al.*, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE SSP*, 2008.

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

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

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

[27] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney, "Model-carrying code: a practical approach for safe execution of untrusted applications," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/945445.945448

[28] Li *et al.*, "Usable Mandatory Integrity Protection For Operating Systems," in *IEEE SSP*, 2007.

[29] W. Sun, R. Sekar, G. Poothia, and T. Karandikar, "Practical proactive integrity protection: A basis for malware defense," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, May 2008.

[30] U. Shankar, T. Jaeger, and R. Sailer, "Toward automated information-flow integrity verification for security-critical applications," in *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium (NDSS'06)*, San Diego, CA, USA, Feb. 2006.

[31] 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*, Oct. 2007, pp. 321–334.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of the* 7^{th} *Symposium on Operating System Design and Implementation*, 2006, pp. 263–278.

[33] D. D. Clark and D. Wilson, "A comparison of military and commercial security policies," in *1987 IEEE Symposium on Security and Privacy*, May 1987.

[34] W. Harris, S. Jha, and T. Reps, "Difc programs by automatic instrumentation," in *Proceedings of Computer and Communications Security (CCS)*, 2010.

[35] D. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–242, 1976.

[36] 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.

[37] S. Hicks, Boniface and, T. Jaeger, and P. McDaniel, "From trusted to secure: building and executing applications that enforce system security," in *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2007.

[38] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers, "Fabric: A platform for secure distributed computation and storage," in *In Proc. ACM Symposium on Operating Systems Principles*, 2009, pp. 321–334.

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

[40] D. E. Bell and L. J. LaPadula, "Secure computer system: Unified exposition and Multics interpretation," Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Hanscom Field, Bedford, MA, Tech. Rep. ESD-TR-75-306, March 1976, also, MITRE Technical Report MTR-2997.

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

[42] D. C. Toll, P. A. Karger, E. R. Palmer, S. K. McIntosh, and S. Weber, "The caernarvon secure embedded operating system," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 1, pp. 32–39, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1341312.1341320

[43] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, September 1975.

[44] 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.