STING: Finding Name Resolution Vulnerabilities in Programs

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

Abstract

The process of *name resolution*, where names are resolved into resource references, is fundamental to computer science, but its use has resulted in several classes of vulnerabilities. These vulnerabilities are difficult for programmers to eliminate because their cause is external to the program: the adversary changes namespace bindings in the system to redirect victim programs to a resource of the adversary's choosing. Researchers have also found that these attacks are very difficult to prevent systematically. Any successful defense must have both knowledge about the system namespace and the program intent to eradicate such attacks. As a result, finding and fixing program vulnerabilities to such as attacks is our best defense. In this paper, we propose the STING test engine, which finds name resolution vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. The key insight is that such name resolution attacks are possible whenever an adversary has write access to a directory shared with the victim, so STING automatically identifies when such directories will be accessed in name resolution to produce test cases that are likely to indicate a true vulnerability if undefended. Using STING, we found 21 previously-unknown vulnerabilities in a variety of Linux programs on Ubuntu and Fedora systems, demonstrating that comprehensive testing for name resolution vulnerabilities is practical.

1 Introduction

The association between names and resources is fundamental to computer science. Using names frees computer programmers from working with physical references to resources, allowing the system to store resources in the way that it sees fit, and enables easy sharing of resources, where different programs may use the different names for the same object. When a program needs access to a resource, it presents a name to a name server, which uses a mechanism called *name resolution* to obtain the corresponding resource.

While name resolution simplifies programming in many ways, its use has also resulted in several types of vulnerabilities that have proven difficult to eliminate. Adversaries may control inputs to the name resolution process, such as namespace bindings, which they can use to redirect victims to resources of the adversaries' choosing. Programmers often fail to prevent such attacks because they fail to validate names correctly, follow adversary-supplied namespace bindings, or lack insight into which resources are accessible to adversaries. Table 1 lists some of the key classes of these vulnerabilities.

While a variety of system defenses for these attacks have been proposed, particularly for name resolution attacks based on race conditions [14, 20, 22, 39, 40, 48, 50–52, 57], researchers have found that such defenses are fundamentally limited by a lack of knowledge about the program [12]. Thus, the programmers' challenge is to find such vulnerabilities before adversaries do. However, finding such vulnerabilities is difficult because the vectors for name resolution attacks are outside the program. Table 1 shows that adversaries may control namespace bindings to redirect victims to privileged resources of their choice, using what we call *improper binding attacks* or redirect victims to resources under the adversaries' control, using what we call *improper resource attacks*. Further, both kinds of attacks may leverage the non-atomicity of various system calls to create races, such as the time-of-check-to-time-of-use (TOCTTOU) attacks [6, 38], which makes them even more difficult for victims to detect.

Researchers have explored the application of dynamic and static analysis to detect namespace resolution attacks. Dynamic analyses [1, 32, 35, 36, 56] log observed system calls to detect possible problems, such as check-

Attack	CWE ID
Improper Binding Attacks	
UNIX Symlink Following	CWE-61
UNIX Hard Link Following	CWE-62
Improper Resource Attacks	
Resource Squatting	CWE-283
Untrusted Search Path	CWE-426
Attacks Caused by Either Bindings or Resources	
TOCTTOU Race Condition	CWE-362

Table 1: Classes of name resolution attacks.

use pairs that may be used in TOCTTOU attacks. However, the existence of problems does not necessarily mean that the program is vulnerable. Many of the check-use pairs found were not exploitable. Static analyses use syntactic analyses [6,53] and/or semantic models of programs to check for security errors [15, 44], sometimes focusing on race conditions [27]. These static analyses do not model the system environment, however, so they often produce many false positives. In addition, several of these analyses result in false negatives as they rely on simpler models of program behavior (e.g., finite state machines), limited alias analysis, and/or manual annotations.

The key insight is that such name resolution attacks are possible only when an adversary has write access to a directory shared with the victim. Using this write access, adversaries can plant files with names used by victims or create bindings to redirect the victim to files of the adversaries' choice. Chari *et al.* [14] demonstrated that when victims use such bindings and files planted by adversaries attacks are possible, so they built a system mechanism to authorize the bindings used in name resolution. However, we find that only a small percentage of name resolutions are really accessible to adversaries and most of those are defended by programs. Further, the solution proposed by Chari *et al.* is prone to false positives, as any pure system solution is, because it lacks information about the programs' expected behaviors [12]. Instead, we propose to test programs for name resolution vulnerabilities by having *the system assume the role of an adversary*, performing modifications that an adversary is capable of, at runtime. Using the access control policy and a list of adversarial subjects, the system can determine whether an adversary has write access to a directory to be used in a name resolution. If so, the system prepares an attack as that adversary would and detect whether the program was exploited or immune to the attack (e.g., did the program follow the symbolic link created?). This is akin to directed black-box testing [23], where a program is injected with a dictionary of commonly known attacker inputs.

In this paper, we design and implement the STING test engine, which finds name resolution vulnerabilities in programs by performing a dynamic analysis of name resolution processing to produce directed test cases whenever an attack may be possible. STING is an extension to a Linux Security Module [58] that implements the additional methods described above to provide comprehensive, system-wide testing for name resolution vulnerabilities. Using STING, we found 21 previously-unknown name resolution vulnerabilities in 19 different programs, ranging from startup scripts to mature programs, such as `cups`, to relatively new programs, such as `x2go`. We detail several bugs to demonstrate the subtle cases that can be found using STING. Tests were done on Ubuntu and Fedora systems, where interestingly some bugs only appeared on one of the two systems because of differences in the access control policies that implied different adversary access.

This research makes the following novel contributions:

- We find that name resolution attacks are always possible whenever a victim resolves a name using a directory where its adversaries have permission to create files and/or links, as defined in Section 3. If a victim uses such a directory in resolving a name, an adversary may redirect them to a resource of the adversary's choosing, compromising victims that use such resources unwittingly.

- We develop a method for generating directed test cases automatically that uses a dynamic analysis to detect when an adversary could redirect a name resolution in Section 4.1.

- We develop a method for system-wide test case processing that detects where victims are vulnerable to name resolution attacks, restores program state to continue testing, and manages the testing coverage in Section 4.2.

- We implement a prototype system STING for Linux 3.2.0 kernel, and run STING on the current versions of Linux distributions, discovering 21 previously-unknown name resolution vulnerabilities in 13 different programs. Perhaps even more importantly, STING finds that 90% of adversary-accessible name resolutions are defended by programs correctly, eliminating many false positives.

We envision that STING could be integrated into system distribution testing to find programs that do not effectively defend themselves from name resolution attacks given that distribution's access control policy before releasing that distribution to the community of users.

2 Problem Definition

Processes frequently require system level resources like files, libraries, and sockets. Since the system's management of these objects is unknown to the process, names are used as convenient references to the desired resource. A *name resolution* server is responsible for converting the requested resource name to the desired object via a *namespace binding*. Typical namespaces in Unix-based systems include the filesystem and System V IPC namespaces (semaphores, shared memory, message queues, etc.). Some namespaces may even support many-to-one mappings (e.g., multiple pathnames may be linked to the same file inode).

Unfortunately, various *name resolution attacks* are possible when an attacker is able to affect this indirection between the desired resource and its name. In this section, we broadly outline two classes of name resolution attacks and give several instances of them. We then discuss how previous efforts attempt to defend against these attacks and their limitations. Finally, we present our solution, STING, that overcomes many of these shortcomings.

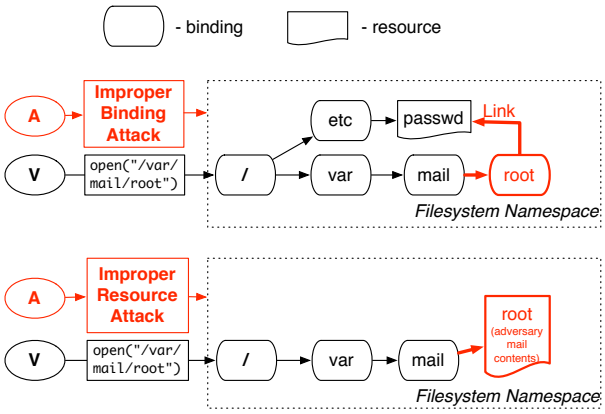


Figure 1: Improper binding and improper resource attacks. A and V are adversary and victim processes respectively.

2.1 Name Resolution Attacks

Malicious parties can control the name resolution process by modifying the namespace's binding to trick victim processes into accessing unintended resources. We find that these attacks can be categorized into two classes. The first, *improper binding attacks*, are when attackers introduce bindings to resources outside of the attackers control. This can give adversary indirect access to the resource through the victim. Such attacks are instances of the confused deputy [33]. The second class, *improper resource attacks*, is when an attacker creates an unexpected binding to a resource the adversary controls.

Instances of these attacks depend on the namespace. For example, the filesystem namespace is often exploited through malicious path bindings like symbolic links and the creation of files with frequently used names. Consider a mail reader program running as `root` attempting to check mail from `/var/mail/root`. Users in the `mail` group are permitted to place files in this directory for the program to read and send. Figure 1 demonstrates how name resolution attacks from both categories could be performed on this program.

- **Symbolic link following:** The adversary wishes to exfiltrate a protected file (`/etc/passwd`) that it cannot normally access. Since users in group `mail` are permitted to create (and delete) bindings (files) in `/var/mail`, the adversary inserts a symbolic link `/var/mail/root` in the namespace that is bound to the desired file. If a victim mail program running as `root` does not check for this link, it might inadvertently leak the protected file. A similar attack can be launched through hard links. This is an instance of an improper binding attack, where adversaries use control of bindings to redirect victim programs with privileges to access or modify resources the adversaries cannot directly.

- **Squatting:** Even if the mail program defends itself against link following attacks, the adversary could simply squat a file on `/var/mail`. If the mail program accepts this file, the adversary could spoof the contents of mail read by `root`. This is an example of an improper resource attack, where the adversary uses control of bindings to create a resource under her control when the victim does not expect to interact with the adversary.

- **Untrusted search path:** Programs frequently rely on files like system libraries or configuration files, but the names they supply to access these files may be wrong. One frequent cause is the program supplying a name relative to its working directory, which causes a problem if the working directory is adversary controlled. Adversaries can then simply bind arbitrary resources at these filenames, possibly gaining control of the victim's program. This is another instance of an improper resource attack, where the adversary supplies an improper resource to the victim.

While the attacks an adversary can carry out are well known, the ways in which programs defend themselves are often ad hoc and complex [13]. Even the most diligent programs may fail to catch all the ways in which an adversary might manipulate these namespaces.

Moreover, defenses to these attacks can often be circumvented through *time-of-check-to-time-of-use (TOCTTOU)* attacks. To do this, the adversary waits until the mail program checks that `/var/mail/root` is a regular file prior to opening it and then switches the file to a link before the `open` call is made. Given the variety of possible name resolution attacks and the complex code needed to defend against them, it should come as little surprise that vulnerabilities of this type continue to be uncovered. Such attacks contribute 5-10% of CVE entries each year.

2.2 Detecting Name Resolution Attacks

Researchers have explored a variety of dynamic and static analyses to detect instances of name resolution attacks, particularly TOCTTOU attacks. However, all such analyses are limited in some ways when applied to the problem of detecting name resolution attacks.

Static Analysis Static analyses of TOCTTOU attacks vary from syntactic analyses specific to *check-use* pairs [6, 53], to building various models of programs to check for security errors [15,44,45], to race conditions in general [27]. However, static analyses are disconnected from essential environmental information, such as the system's access control policy to determine whether an adversary can even launch an attack. For example, a program may legitimately accesses files in `/proc` without checking for name resolution attacks; however, the same cannot be done in `/tmp`. Thus, these analyses yield a significant number of false positives. Further, static techniques are limited to TOCTTOU attacks, due to the absence of standardized program checks against name resolution attacks in general.

Dynamic Analysis Dynamic analyses [1,32,35,36,56] typically take a system-wide view, logging observed system calls from processes to detect possible problems, such as check-use pairs. Dynamic analyses can also detect specific vulnerabilities, either at runtime [36] or after the fact [35]. Compared to static analyses, dynamic analyses can take into account the system's environment, but suffer the disadvantage of being unaware of the internal code of the program. In addition, the quality of dynamic analysis is strongly dependent on the test cases produced. Because name resolution attacks require an active adversary, the problem is to produce adversary actions in a comprehensive manner. Using benign system traces may identify some vulnerabilities, such as those built-in to the normal system configuration [13], but will miss many other feasible attacks. Finally, any dynamic analysis must distinguish program actions that are safe from those that are vulnerable effectively. We have found that programs successfully defend themselves from a large percentage of the attempted name resolution attacks (only 12.5% were vulnerable), so test case processing must find cases where program defenses are actually missing or fail. Since previous dynamic analyses lack insight into the program, several false positives have resulted.

Symbolic Execution Researchers have recently had success finding the conditions under which a program is vulnerable using *symbolic execution* [7, 8, 10, 11, 18, 19, 25, 29–31, 46]. Symbolic execution has been used to produce test cases for programs to look for bugs [9, 37], to generate filters automatically [8, 18], and to generate test cases to leverage vulnerabilities in programs [3] automatically. In these symbolic execution analyses, the program is analyzed to find constraints on the input values that lead to a program instruction of interest (i.e., where an error occurs). Then, the symbolic execution engine solves for those constraints to produce a concrete test case that when executed would follow the same path. Finding name resolution attacks using symbolic execution may be difficult because the conditions for attack are determined mainly by the operating environment rather than the program. While symbolic execution often requires a model of the environment in which to examine the program, the environment needs to be the focus of analysis for finding name resolution attacks.

2.3 Our Solution

As a result, we use a dynamic analysis to find name resolution vulnerabilities, but propose four key enhancements to overcome the limitations of prior analyses of all types.

First, each name resolution system call is evaluated at runtime to find the bindings used in resolution and to determine whether an adversary is capable of applying one or more of the attack types listed in Table 1. If so, a test case resource is automatically produced at an adversary-redirected location in the namespace and provided to the victim. As a result, test cases are only applied where adversaries have the access necessary to perform such attacks.

Second, we track the victim's use of the test case resource to determine whether it accepts the resource as legitimate. If the victim uses the resource (e.g., reads from it), we log the program entrypoint[1] that obtained the resource as vulnerable. While it is not always possible to exploit such a flaw to compromise the program, this approach greatly narrows the number of false positives while still locating several previously-unknown true

[1] A program entrypoint is a program instruction that invoked the name resolution system call, typically indirectly via a library (e.g., libc).

vulnerabilities. We also log the test cases run by program entrypoint to avoid repeating the same attack.

Third, another problem with dynamic analysis is ensuring that the analysis runs comprehensively even though programs may fail or take countermeasures when attacks are detected. We take steps to keep programs running regardless of whether they fall victim to the attack or not. Our test case resources use the same data as the expected resource to enable vulnerable programs to continue, and we automatically revert namespaces after completion of a test and restart programs that terminate when an attempted attack on them is detected.

3 Testing Model

In this section, we define an adversarial model that we use to generate test cases that can be used to identify program vulnerabilities.

Our goal is to discover vulnerabilities that will compromise the *integrity* of a process. Classically, an integrity violation is said occur when a lower integrity process provides input to a higher integrity process or object [5, 16]. For the name resolution attacks described in the last section (see Table 1), integrity violations are created in two ways: (1) improper binding attacks, where adversaries may redirect name resolutions to resources that are normally not modifiable by adversaries, enabling adversaries to modify higher integrity objects, and (2) improper resource attacks, where adversaries may redirect name resolutions to resources that are under the adversaries' control, enabling adversaries to deliver lower integrity objects to processes. In this section, we define how such attacks are run and detected to identify the requirements for the dynamic analysis.

A nameserver performs namespace resolution by using a sequence of namespace bindings, $b_{ij} = (r_i, n_j, r_k)$, to retrieve resource r_k from resource r_i given a name n_j. In a file system, r_i is a directory, n_j is an element of the name supplied to the nameserver for resolution, and r_k is another directory or a file. Attacks are possible when adversaries of a victim program have access to modify binding b_{ij} to $(r_i, n_j, r_{k'})$ or create such a binding if it does not exist, enabling them to redirect the victim's process to a resource r'_k instead of r_k. Since bindings cannot be modified like files, adversaries generally require the delete permission to remove the old binding and the create permission to create the desired binding to perform such modification. Two types of name resolution attacks are possible when adversaries have such permission (e.g., write permission to directories in UNIX systems).

Improper binding attacks use the permission to modify a binding to create a link (symbolic or hard) to an existing resource that is inaccessible to the adversary, as in symbolic and hard link attacks described above. That is, the improper binding may lead to privilege escalation for the adversary by redirecting the victim process to use an existing resource on behalf of that adversary.

Improper resource attacks use the permission to modify a binding to create a new resource controlled by the adversary. That is, the adversary tries to trick the victim into using the improper resource to enable the adversary to provide malicious input to the victim, such as in resource squatting and untrusted search path attacks described above.

STING discovers name resolution vulnerabilities by identifying scenarios where an attack is possible and generating test cases to validate the vulnerability. Whenever a *name resolution system call* is requested by the victim (i.e., a system call that converts a name to a resource reference, such as `open`), STING finds the bindings that would be used in the namespace resolution process to determine whether an adversary of the process has access to modify one or more of these bindings. If so, STING generates an *attack test case* by producing a test case resource, which emulates either an existing, privileged resource or a new adversary-controlled resource, and adjusting the bindings as necessary to resolve to that resource. A reference to this test case resource is returned to the victim.

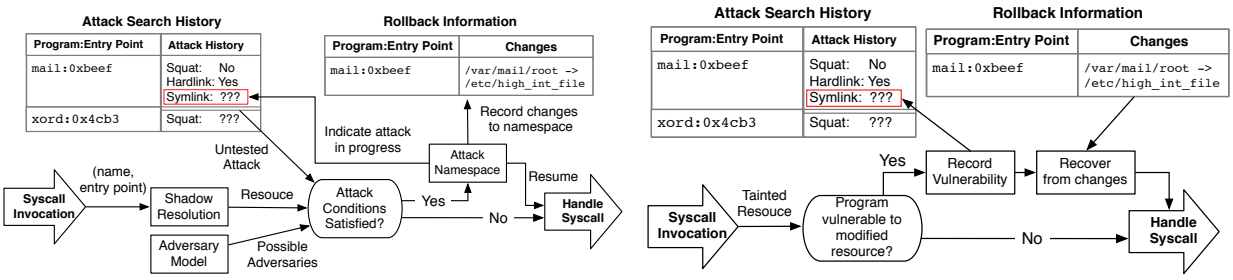
Vulnerability in the Victim. We define a victim to be vulnerable if the victim runs an *accept system call* using a reference to the test case resource.

A victim accepts a test case resource if it runs an *accept system call*, a system call that uses the returned reference to the test case resource to access resource data (e.g., `read` or `write`). If a victim is tricked into reading or writing a resource inaccessible to the adversary, the adversary can modify the resource illicitly[2]. If a victim is tricked into reading or writing a resource that is controlled by the adversary, then the adversary can control the victim's processing.

4 Design

The design of STING is shown in Figure 2. STING is divided into two phases. The *attack phase* of STING is invoked at the start of a name resolution system call. STING resolves the name itself to obtain the bindings that would be used in normal resolution, and then determines whether an attack is possible using the program's adversary model. When an attack is possible, STING chooses an attack from the list in Table 1 that has not already been tried and produces a test case resource and asso-

[2] A read operation on a test case resource is indicative of integrity problems if the resource is opened with read-write access.



(a) Launching an attack

(b) Detecting a vulnerability

Figure 2: STING consists of two phases: (a) launching an attack, and (b) detecting a vulnerability due to the attack and recovering original namespace state.

ciated bindings to launch the attack. The *detect phase* of STING is invoked on accept system calls. This phase detects if a process "accepted" STING's changes, indicating a vulnerability, and records the vulnerability information in the previously added entry in the attack history. STING reverts the namespace to avoid side-effects. These two phases are detailed below.

STING is designed to test systems, not just individual programs, so STING will generate test cases for any program in the system that has an adversary model should the opportunity arise. To control the environment under which a program is run, STING intercepts `execve` system calls. For example, programs that may be run by unprivileged users (e.g., setuid programs) are started in an adversary's home directory by this interception code. Other than initialization, the attack and detect phases are the same for all processes.

4.1 Attack Phase

Shadow resolution. Whenever a name resolution system call is performed, STING needs identify whether an attack is possible against that system call. The first step is to collect the bindings that would normally be used in the resolution. We cannot use the existing name resolution mechanism, however, since that has side-effects that may impact the process and also does not gather the desired bindings for evaluation. Instead, we perform a *shadow resolution* that only collects the bindings.

There are two challenges with shadow resolution. First, we have to ensure that all name resolution operations performed by the system are captured in the shadow resolution. This task can be tricky because some name resolution is performed indirectly. For example, `exec` resolves the interpreter that executes the program in the "shebang" line in addition to the program whose name is an argument to the system call. To capture all the name resolution code, we use Cscope [3] to find all the system calls that invoke a fundamental function of name resolution, `do_path_lookup`. Using this we find 62 system calls that do name resolution for the Linux filesystem. The three System V IPC system calls that do name resolution were identified manually.

Second, we need to modify the name resolution code to collect the bindings used without causing side-effects in the system. Fortunately, the name resolution code in Linux does not cause side-effects itself. The system call code that uses name resolution creates the side-effects. Thus, we simply invoke the name resolution functions directly when the system call is received. Some effort must be taken to format the call to the name resolution code at the start of the system call, but fortunately the necessary information is available (name, flags, etc.).

Find vulnerable bindings. To carry out an attack, STING has to determine whether any adversary of the program has the necessary permissions to the bindings involved in the resolution. To answer this question, we need to identify the program's adversaries and evaluate the permissions these adversaries have to bindings efficiently. We note that the specific permissions necessary to launch an attack are specified in Section 3.

We do not want the dynamic analysis to depend on a single adversary model for the system, but instead permit the use of *program-specific adversary models*. The adversaries of a process are determined by the process's subject (i.e., in the access control policy) and optional program-specific sets of subjects and/or objects that are adversaries or adversary-controlled, respectively. From this information, a comprehensive set of adversary subjects are computed. Using a discretionary access control (DAC) policy, an adversary is any subject other than the victim and the trusted superuser `root`. Chari *et al.* used the DAC policy in their dynamic analysis [13], which worked adequately for `root` processes but incurred some false positives for processes run under other subjects. For systems that enforce a mandatory access control (MAC) policy, methods have been devised to compute the adver-

[3] http://cscope.sourceforge.net/

saries of individual subjects [34, 49]. We note that MAC adversaries may potentially be running processes under the same DAC user, so they are typically finer-grained.

Finding the permissions of a process's adversaries at runtime must be done efficiently. If a process has several adversaries, the naive technique of querying the access control policy for each adversary in turn is unacceptable. To solve this, we observe we can precompute the adversaries of particular process as in a capability-list, where each process has a list of tuples consisting of an object (or label in a MAC policy), a list of adversaries with create permission to that object (or label), and the list of adversaries with delete permission to that object (or label). We store these in a hash table for quick lookup at runtime.

Identify possible attacks. Once we identify a binding that is accessible to an adversary, we need to choose an attack from which to produce a test case. For improper binding attacks, an attack needs to modify a binding from an existing resource to the target resource using a symbolic or hard link. Such attacks are only possible in the Linux filesystem namespace, where a single file (inode) may have multiple names.

Improper resource attacks are applicable across all namespaces. We consider two instances of improper resource attacks (see Table 1). For resource squatting, attacks are only meaningful if the adversary can supply a resource with a lower integrity than the victim intended to access. To determine the victim's intent, we simply check if a non-adversarial subject has permissions to supply the resource the adversary is attacking [4]. This occurs in directories shared by parties at more than one integrity level. If so, we assume that the victim intended to access the higher integrity file (i.e., one that could be created by a non-adversarial subject), and attempt a squatting attack which succeeds if the victim later accepts the test case resource. MOPS [44] uses a similar but narrower heuristic to identify intent and detect ownership stealing attacks, which are another case of resource squatting attacks.

Launch an attack. Launching an attack involves making modifications to the namespace to generate realistic attack scenarios. Different attacks modify the namespace in different ways. For improper binding attacks, we create a new test case resource (e.g., file) that represents a privileged resource, and change the adversary-modifiable bindings to point to it (e.g., symbolic link). For improper resource attacks, we replace the existing resource (if present) with a new test case resource and binding.

Modification of the filesystem namespace in particular presents challenges of backing up existing files, rollback and multiple views for different subjects. First, we have to change the file system to create the test cases, such as deleting existing files. Second, once the test case finishes, we need to rollback the namespace to its original state. While we can back up files (costing the overhead of copy), other resources such as UNIX domain sockets are hard or impossible to rollback once destroyed. Another requirement is that the attack should only be visible to the appropriate victim subjects having the attacker as an adversary. Thus, direct modification of the existing filesystem is undesirable.

To solve the above problems, we take inspiration from the related problem of filesystem unions. Union filesystems unite two or more filesystems (called branches) together [2, 59]. A common use-case is in Live CD images, where the base filesystem mounted from a CDROM is read-only, and a read-write branch (possibly temporary) is mounted on top to allow changes. When a file on the lower branch is modified, it is "copied up" to the upper branch and thereafter hides the corresponding lower branch file. "Whiteouts" are created on the upper branch for deleted files.

To support STING, our general strategy is thus to have a throw-away "upper branch" mounted on top of the underlying filesystem "lower branch". STING creates resources only on the upper branch. As STING does not deal with data, files are created empty. Next, STING directs operations to upper branches if the resource exists on the upper branch *and* was created by an adversary to the currently running process. This enables different processes to have different views of the filesystem namespace.

Once a test case resource is created, we taint it using extended attributes to identify when it is used in an accept system call[5], signaling a vulnerability. We also record rollback information about the resources created in a rollback table.

These changes to the bindings have to be done as the adversary. The most straightforward option is to have a separate "adversary process" that is scheduled in to perform needed changes. This was the first option we explored; however, it introduced significant performance degradation due to scheduling. Instead, we perform the change using the currently running victim process itself. We change the credentials of the victim process to that of the adversary, carry out the change, and then revert to the victim's original credentials. We do this without leaving behind side effects on the victim process' state. For example, if we create a namespace binding using `open`, we close the opened file descriptor.

[4] We discount `root` superuser permissions while checking non-adversarial subjects, as otherwise root will be a non-adversary in any directory.

[5] Some filesystems do not support extended attributes. Since we use tmpfs as the upper branch, we extended it to add such support for our testing. For other namespaces such as IPCs, we store the taint status in a field in the `security` data structure defined by SELinux.

There are some cases where the system needs to revert a test case resource back to a "benign" version. "Check" system calls [56] (e.g., `stat, lstat`) resolve the name to verify properties of the resource, so attacks should present a benign resource to prevent the victim from detecting the attack. We simply redirect such accesses to the lower branch.

In addition to adversarial modification of the namespace, STING also changes process attributes relevant to name resolution. In particular, it changes the working directory a process is launched in to an adversary-controlled directory.

We want to prevent STING from launching the same attack multiple times. Trying the same attack on the same system call prevents forward exploration of the attack space and further program code from being exercised. A unique attack associates an attack type with a particular system call entrypoint in the program. Thus, when we launch an attack, we check an attack history that stores which attack types have been attempted already and their result (see Figure 2). We do not attempt multiple binding changes for an attack type. We have not found any programs that perform different checks for name resolution attacks based on the names used. Tracking such history requires unique identification of system call entrypoints in the program, which we discuss in Recording below.

4.2 Detect Phase

Detect vulnerability. Detecting whether a victim is vulnerable to an attack is relatively straightforward – we simply have to determine if the program "accepted" the test case resource. Definition of acceptance for different attacks are presented in Table 2. On the other hand, we conclude that the program defends itself properly from an attack if it: (1) exits without accepting the test case resource or (2) retries the operation at the same program entrypoint. When detection determines that a victim is vulnerable or invulnerable, it fills this information in the attack history entry created during the attack phase, and optionally logs the fact.

STING detects successful attacks by identifying use of a test case resource. Each test case resource is marked when returned to the victim. To detect when a victim uses a test case resource, we must have access to the inode, so such checking is integrated with the access control mechanism (e.g., Linux Security Module). Once a test case resource is found, we need to determine if it is being accepted by retrieving the system call invoked. As an access control check may apply to multiple system calls, we have to retrieve the identity of the system call from the state of the calling process. Vulnerabilities found have their attack history record logged into user space.

Attack	Accept
Symbolic link	write-like, `read`, `readlink`
Hard link	write-like, `read`
File squat	write-like, `read`
UNIX-domain socket squat	`connect`
System V IPC squat	`msgsnd`, `semop`, `shmat`

Table 2: Table showing calls that signify acceptance, and therefore detection, for various attacks. write-like system calls are any calls that modifies the resource's data or metadata (e.g., fchmod).

We note that the process of detecting a vulnerability is the same for all attack types, including those based on race conditions. STING automatically switches resources between check and use as discussed above, so we only need to detect when an untrusted resource is accepted. `fstat` is not an accept system call, so the "use" of the test case resource in that system call does not indicate a vulnerability. Thus, if the program should somehow detect an attack using `fstat`, preventing further use of the test case resource, then STING will not record a vulnerability.

Update attack history. Once a particular attack has been tried on a system call, trying it again in future invocations of the program is redundant and may prevent further code from being exercised. Avoiding this problem requires storing attacks tested for system calls in the attack history. The challenge is unique identification of the system call entrypoint, which uniquely identifies the instruction from which the program made the system call. To find this instruction, we perform a backtrace of the user stack to find the first frame within the program that is not in the libraries. We also extend our system to support interpreters by porting interpreter debugging code into the kernel that locates and parses interpreter data structures to the current line number in the script, for the Bash, PHP and Python interpreters. Only between 11 and 59 lines of code were necessary for each interpreter. We use the current line number in the script as the entrypoint for interpreted programs.

Namespace recovery. Finally, we make changes so that STING can work online despite changing the namespace state. While it appears that such changes could cause processes to crash, we have not found this to be the case. Unlike data fuzzing, we find changes in namespace state do not cause programs to arbitrarily crash, as we preserve data and only change resource metadata. When an attack succeeds, the only change needed is to redirect the access to the corresponding resource in the lower branch of the unioned filesystem that contains the actual data (if one exists), and delete the resource in the upper branch. On the other hand, if the attack fails, STING again deletes the resource in the up-

per branch. Programs that protect themselves proceed in two ways. First, the program might retry the same system call again. Interestingly, we find this happens in a few programs (Section 6.2). In this case, STING will not launch an attack at that entrypoint again, and the program again continues. Second, the program might exit. If so, STING records that the attack failed at that entrypoint and restarts the program with its original arguments (recorded via `execve` interception). For many programs that exit, restarting them from the beginning does not affect system correctness. Thus, we find our tool can perform online without complex logic. We are currently exploring how to integrate process checkpointing and rollback [17] to carry out recovery more gracefully for the exit cases.

5 Implementation

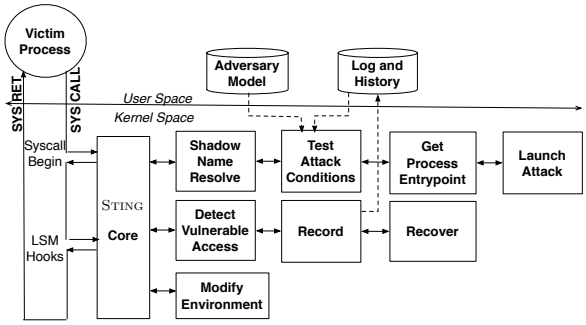


Figure 3: STING is implemented in the Linux kernel and hooks on to the system call beginning and LSM hooks. It has a modular implementation. We show here in particular the interaction between user and kernel space.

Figure 3 shows the overall implementation of STING . STING is implemented in the Linux 3.2.0 kernel. It hooks into LSM through the SELinux function `avc_has_perm` for its detect phase, and into the beginning of system calls for its attack phase. STING has an extensible architecture, with modules registering themselves, and rules specifying the order and conditions under which modules are called.

The modules implement functionality corresponding to the steps shown in the design (Figure 2). The entrypoint module uses the process' `eh_frame` ELF section to perform the user stack unwinding. `eh_frame` replaces the DWARF `debug_frame` section, and is enabled by default on Ubuntu 11.10 and Fedora 15 systems. Stack traces in interpreters yield an entrypoint inside the interpreter, and not the currently executing script. We extended our entrypoint module to identify line numbers inside scripts for the Bash, PHP and Python interpreters [54].

Server Programs Installed	
BIND DNS Server	Apache Web Server
MySQL Database	PHP
Postfix Mail Server	Postgresql Database
Samba File Server	Tomcat Java Server

Table 3: Server programs installed on Ubuntu and Fedora.

The data for the MAC adversary model is pushed into the kernel through a special file, and code to use each of these to decide adversary access is in the test attack conditions module. After launching an attack, the modified resources are tainted through extended attributes for later detection when a victim program uses the resource. We had to extend the `tmpfs` filesystem to handle extended attributes. The recording module can print vulnerabilities as they are detected, and also log the vulnerabilities and search history into userspace files through `relayfs`. A startup script loads the attack search history into the kernel during bootup, so the same attacks are not tried again.

We prototyped a modified version of the UnionFS filesystem [59] for STING. We mount a `tmpfs` as the upper branch, and the root filesystem as the lower branch. The main change involved redirecting to the upper or lower branches depending on a subject's adversaries, and disabling irrelevant UnionFS features such as copy-up.

6 Evaluation

We first evaluate STING 's ability to finding bugs, as well as broader security issues in Section 6.1. We then analyze the suitability of STING as an online testing tool in Section 6.2

6.1 Security Evaluation

The aim of the security evaluation is to show that:

- STING can detect real vulnerabilities with a high percentage of them being exploitable in both newer programs, and older, more mature programs, and
- Normal runtime and static analysis would result in a large number of false positives, and normal runtime would also miss some attacks.

We tested STING on the latest available versions of two popular distributions - Ubuntu 11.10 and Fedora 16. In both cases, we installed the default base Desktop distribution, and augmented them with various common server programs (Table 3). Note that STING requires no additional special setup; it simply reacts to normal name resolution requests at runtime. We collected data on both systems over a few days of normal usage.

Adversary model	Total Resolutions	Adversary Access	Vulnerable
DAC - Ubuntu	2345	134 (5.7%)	21 (0.9%)
DAC - Fedora	1654	66 (4%)	5 (0.3%)

Table 4: Table showing the total number of distinct entrypoints invoking system calls performing namespace resolutions, number accessible to adversaries under an adversary model, and number of interfaces for which STING detected vulnerabilities.

Type of vulnerability	Total
Symlink following	22
Hardlink following	14
File squatting	10
Untrusted search	6
Race conditions	7

Table 5: Number and types of vulnerabilities we found. Race is the number of TOCTTOU vulnerabilities, where a check is made but the use is improper. A single entrypoint in Table 6 may be vulnerable to more than one kind of attack.

Program	**Vuln. Entry**	**Priv. Escalation DAC: uid->uid**	**Distribution**	**Previously known**
dbus-daemon	2	messagebus->root	Ubuntu	Unknown
landscape	4	landscape->root	Ubuntu	Unknown
Startup scripts (3)	4	various->root	Ubuntu	Unknown
mysql	2	mysql->root	Ubuntu	1 Known
mysql_upgrade	1	mysql->root	Ubuntu	Unknown
tomcat script	2	tomcat6->root	Ubuntu	Known
lightdm	1	*->root	Ubuntu	Unknown
bluetooth-applet	1	*->user	Ubuntu	Unknown
java (openjdk)	1	*->user	Both	Known
zeitgeist-daemon	1	*->user	Both	Unknown
mountall	1	*->root	Ubuntu	Unknown
mailutils	1	mail->root	Ubuntu	Unknown
bsd-mailx	1	mail->root	Fedora	Unknown
cupsd	1	cups->root	Fedora	Known
abrt-server	1	abrt->root	Fedora	Unknown
yum	1	sync->root	Fedora	Unknown
x2gostartagent	1	*->user	Extra	Unknown
19 Programs	**26**			**21 Unknown**

Table 6: Number of vulnerable entrypoints we found in various programs, and the privilege escalation that the bugs would provide.

6.1.1 Finding Vulnerabilities

Using a DAC attacker model, in total, STING found 26 distinct vulnerable resolutions across 20 distinct programs (including scripts). Of the 26 vulnerable resolutions, 5 correspond to problems already known but unfixed. 17 of these vulnerabilities are *latent* [13], meaning a normal local user would have to gain privileges of some other user and can then attempt further attacks. For example, one bug we found required the privileges of the user `postgres` to carry out a further attack on `root`. This can be achieved, for example, by remote network attackers compromising the PostgreSQL daemon. For all vulnerabilities found, we manually verified the source code that a name resolution vulnerability existed. Several bugs were reported, of which 2 were deemed not exploitable (although a name resolution vulnerability existed) (Section 6.1.3).

Table 4 shows the total number of distinct name resolutions received by STING that were attackable . This data shows challenges facing static and normal runtime analysis. Only 4-5.7% of the total name resolutions are accessible to the adversary under the DAC adversary model. Therefore, static analysis that looks at the program alone will have a large number of false positives, because programs do not have to protect themselves from name resolutions inaccessible to the adversary. Second, normal runtime analysis cannot differentiate between when programs are vulnerable and when they protect themselves appropriately. We found $< 1\%$ of the name resolutions accessible to the adversary are actually vulnerable to different name resolution attacks. Further, 6 of these vulnerabilities would simply not have been uncovered during normal runtime; they are untrusted search paths they require programs to be launched in insecure directories.

Table 5 shows the total number of vulnerabilities by type. A single entrypoint may be vulnerable to more than one type of attack. We note that STING was able to find vulnerabilities of all types, including 7 that required race conditions.

Table 6 shows the various programs across which vulnerabilities were found. Interestingly, we note that 6 of the 24 vulnerable name resolutions in Ubuntu were found in Ubuntu-specific scripts. For example, CVE-2011-4406 and CVE-2011-3151 were assigned to two bugs in Ubuntu-specific scripts that STING found. Further, the programs containing vulnerabilities range from mature (e.g., `cupsd`) to new (e.g., `x2go`). We thus believe that STING can help in detecting vulnerabilities before an adversary, if run on test environments before they are deployed.

MAC adversary model. We carried out similar experiments for a MAC adversary model on Fedora 16's default SELinux policy. We assume an adversary limited only by the MAC labels, and allow the adversary permissions to run as the same DAC user. This is one of the aims of SELinux – even if a network daemon running as `root` gets compromised, it should still not compromise the whole system arbitrarily. However, we found that the SELinux policy allowed subjects we consider untrusted (such as the network-facing daemon `sendmail_t`) create permissions to critical labels such as `etc_t`. Thus STING immediately started reporting vulnerable name resolutions whenever any program accessed `/etc`. Thus, either the SELinux policy has to be made stricter, the adversary model must be weakened for mutual trust among all these programs, or all programs have to defend themselves from name resolution attacks in `/etc` (which is probably impractical). This problem is consistent with the findings that `/etc` requires exceptional trust in the SELinux policy reported elsewhere [42].

```
01 /* filename = /var/mail/root */
02 /* First, check if file already exists */
03 fd = open (filename, flg);
04 if (fd == -1) {
05     /* Create the file */
06    fd = open(filename, O_CREAT|O_EXCL);
07    if (fd < 0) {
08        return errno;
09    }
10 }
11 /* We now have a file. Make sure
12 we did not open a symlink. */
13 struct stat fdbuf, filebuf;
14 if (fstat (fd, &fdbuf) == -1)
15     return errno;
16 if (lstat (filename, &filebuf) == -1)
17     return errno;
18 /* Now check if file and fd reference the same file,
19    file only has one link, file is plain file.  */
20 if ((fdbuf.st_dev != filebuf.st_dev
21     || fdbuf.st_ino != filebuf.st_ino
22     || fdbuf.st_nlink != 1
23     || filebuf.st_nlink != 1
24     || (fdbuf.st_mode & S_IFMT) != S_IFREG)) {
25     error (_("%s must be a plain file
26         with one link"), filename);
27     close (fd);
28     return EINVAL;
29 }
30 /* If we get here, all checks passed.
31    Start using the file */
32 read(fd, ...)
```

Figure 4: Code from the GNU mail program in `mailutils` illustrating a squat vulnerability that STING found.

6.1.2 Examples

In this section, we present particular examples highlighting STING's usefulness, and also broader lessons.

Mail Programs. GNU `mail` is the default mail client on Ubuntu 11.10, in which STING found a vulnerability. This example shows the difficulty of proper checking in programs, and why detection tools with low false positives are necessary – programmers can easily get such checks wrong, and there are no standardized ways to write code to defend against various name resolution attacks.

The code shows the program preparing to read the file `/var/mail/root`. In summary, this program creates an empty file when the file doesn't already exist (lines 4-10), using flags (`O_EXCL`) to ensure that a fresh file is created. The program performs several checks to verify the safety of the file opened, guarding against race conditions and link traversal (both symbolic and hard links) (11-29). Unfortunately, the program fails to protect itself against a squatting attack if the file already exists, as it does not check `st_uid` or `st_gid`; any user in group `mail` can control the contents of `root`'s inbox. Interestingly, it protects itself against squatting attacks on line 6.

X11 script STING found a race condition exploitable by a symbolic link attack on the script that creates `/tmp/.X11-unix` in Ubuntu 11.10. The code snippet is shown in Figure 5. The aim of the script is to create a root-owned directory `/tmp/.X11-unix`. Lines 6-8 check if such a file already exists that is not a directory, and if so, moves it away so a directory can be created. In Line 9, the programmer creates the directory, and assumes it will succeed, because the previous code had just moved any file that might exist. However, because `/tmp` is a shared directory, an adversary scheduled in between the moving of the file and the `mkdir` might again create a file in `/tmp/.X11-unix`, thus breaking the programmer's expectation. If the file is a link pointing to, for example, `/etc/shadow`, the `chmod` on Line 11 will make it world-readable. STING was able to detect this race condition by changing the resource into a symbolic link after the move and before the creation on line 9, as it acts just before the system call on line 9. This script has existed for many years, showing how it is easy to overlook such conditions. This also shows how STING can synchronously produce any race condition an adversary can, because it is in the system. This script was independently fixed by Ubuntu in its latest release. The discussion page for the bug [21] shows how such checks are challenging to get right even for experienced programmers. Consequently, manually scanning source code can also easily miss such vulnerabilities.

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

Figure 5: Code from an X11 startup script in Ubuntu 11.10 that illustrates a race condition that STING found.

mountall This program has an untrusted search path that is not executed in normal runtime but was discovered by STING. This Ubuntu-specific utility simply mounts all filesystems in `/etc/fstab`. When launched in an untrusted directory, it issues mount commands that search for files such as `none` and `fusectl` in the current working directory. If these are symbolic links, then the contents of these files are read through `readlink`, and put in `/etc/mtab`. Thus, the attacker can influence `/etc/mtab` entries and potentially confuse utilities that depend on this file, such as unmounters. This is an example of how very specific conditions are required to detect the attack – the program needs to be launched in an adversarial directory, and the name searched for needs to be a symbolic link. Normal runtime would not give any hint of such attacks.

postgresql init script. This vulnerability highlights the challenge facing developers and OS distributors. This script runs as `root`, and is vulnerable to symbolic and hard link attacks on accessing files in `/etc/postgresql`. That directory is owned by the user `postgres`, which could be compromised by remote attacks on PostgreSQL, who can then use this vulnerability to gain `root` privileges. The problem is that the developers who wrote the script did not expect the directory `/etc/postgresql` to be owned by a non-root user. However, the access control policy did not reflect this assumption. STING is useful in finding such discrepancies in access control policies as it can run with attacker models based on different policies.

6.1.3 False Positives

Two issues in STING cause false positives.

Random Name. The programs `yum, abrt-server, zeitgeist-daemon` in Table 6 were vulnerable to name resolution attacks, but defended themselves by creating files with random names. Library calls such as `mktemp` are used to create such names. STING cannot currently differentiate between "random" and "non-random" names. Exploiting such vulnerabilities requires the adversary to guess the filename, which may be challenging given proper randomness. In any case, such bugs can be fixed by adding the `O_EXCL` flag to `open` when creating files.

Program Internals. STING does not know the internal workings of a program. Thus, it cannot know if use of a resource from a vulnerable name resolution can affect the program or not, and simply marks it for further perusal. A vulnerable name resolution involving a write-like accept operation can always be exploited. However, whether those involving `read` can be exploited depends on the internals of the program. Eight of the 26 vulnerable name resolutions in Table 6 are due to `read`. While this has led to some false positives (two additional vulnerable name resolutions involving `read` not in Table 6 were deemed to not affect program functioning), STING narrows the programmers' effort significantly. Nonetheless, more knowledge regarding program internals would improve the accuracy even further.

6.2 Performance Evaluation

We measured the performance of STING to assess its suitability as an online testing tool. While the performance of STING is of not of primary importance because it is meant to be run on test environments in non-production systems before deployment, it must nevertheless be responsive to online testing. We measured performance using micro- and macro-benchmarks. While STING does cause noticeable overhead, it did not impede our testing. All tests were done on a Dell Optiplex 980 machine with 8GB of RAM.

Case	Time (μs)	Overhead
Attack Phase: `open` system call		
Base	14.57	–
+ Find Vulnerable Bindings	31.44	2.15×
+ Obtain entrypoint and check attack history	211.20	12.33×
+ Launch attack	365.87	25.1×
Detect Phase: `read` system call		
Base	8.73	–
+ Detect vulnerability	9.18	1.05×
+ Namespace recovery	63.08	7.22×

Table 7: Micro-overheads for system calls showing median over 10000 system call runs.

Benchmark	Base	STING	Overhead
Apache 2.2.20 compile	151.65s	163.79s	8%
Apachebench: Throughput	231.77Kbps	221.89Kbps	4.33%
Latency	1.943ms	2.088ms	7.46%

Table 8: Macro-benchmarks showing compilation and Apachebench throughput and latency overhead. The standard deviation was less than 3% of the mean.

Micro-performance (Table 7) was measured by the time taken for individual system calls under varying conditions. For an attack launch, system call overhead is caused by the time to (1) detect adversary accessibility, (2) get and compare process entrypoint against attack history, and (3) launch the attack. The main overhead is due to obtaining the entrypoint to check the attack history and carrying out the attack. However, obtaining the entrypoint is required only if the name resolution is adversary accessible (around 4-5.7% in Table 4), and an attack is launched only once for a particular entrypoint, thereby alleviating their impact on overall performance. For the detection phase, we have (1) detect vulnerable access, and (2) rollback namespace. Namespace recovery through rollback is expensive, but occurs only once per attack launched.

Macro-benchmarks (Table 8) showed upto 8% overhead. Apache compilation involved a lot of name resolutions and temporary file creation. During our system tests, the system remained responsive enough to carry out normal tasks, such as browsing the Internet using Firefox and checking e-mail. We are investigating further opportunities to improve performance.

Program retries and restarts. We came across thirteen programs that retried a name resolution system call on failure due to a STING attack test case. The most

common case was temporary file creation – programs retry until they successfully create a temporary file with a random name. Programs that retry integrate well with STING , which maintains its attack history and does not retry the same attacks on the same entrypoints. On the other hand, a few programs exited on encountering an attack by STING . We currently simply restart such programs (Section 4.2). For example, `dbus-daemon` exited during boot due to a STING test case and had to be restarted by STING to continue normal boot. However, programs may lose state across restarts. We are investigating integrating process checkpoint and rollback mechanisms [17].

7 Related Work

Related work that deals with detecting name resolution attacks was presented in Section 2.2. Here, we discuss dynamic techniques to detect other types of program bugs, and revisit some dynamic techniques that detect name resolution attacks.

Black-box testing. Fuzz testing, an instance of black-box testing, runs a program under various input data test cases to see if the program crashes or exhibits undesired behavior. Particular program entrypoints (usually network or file input) are fed totally random input with the hope of locating input filtering vulnerabilities such as buffer overflows [24, 41, 47]. Black-box fuzzing generally does not scale because it is not directed. To alleviate this, techniques use some knowledge of the semantics of data expected at program entrypoints. SNOOZE [4] is a tool to generate fuzzing scenarios for network protocols using which bugs in programs were discovered. TaintScope [55] is a directed fuzzing tool that generates fuzzed input to pass checksum code in programs. Web application vulnerability scanners [23] supply very specific strings to detect XSS and SQL injection attacks.

We find a parallel can be drawn between our approach and directed black-box testing, where semantics of input data is known. While such techniques change the *data* presented to a program to exercise program paths with possible vulnerabilities, we change the resource, or the *metadata* presented to the application for the same purpose. Thus, STING can be viewed as an instance of black-box testing that changes the namespace state to evaluate program responses.

Taint Tracking. Taint techniques track flow of tainted data inside programs. They can be used to find bugs given specifications [60], or can detect secrecy leaks in programs [26]. Flax [43] uses a taint-enhanced fuzzing approach to detect input filtering vulnerabilities in web applications. However, taint tracking by itself does not actively change any data presented to applications, and thus has different applications.

Dynamic Name Resolution Attacks Detection. As mentioned in Section 2.2, most dynamic analysis are specific to detecting TOCTTOU attacks. Chari *et al.* [13] present an approach to defend against improper binding attacks; however, they cannot detect them until they actively occur in the system. We use active adversaries to generate test cases and to exercise potentially vulnerable paths in programs to detect vulnerabilities that would not occur in normal runtime traces. Further, none of the approaches deal with improper resource attacks, of which we detect several.

8 Discussion

Other System State Attacks. More program vulnerabilities may be detected by modifying system state. For example, non-reentrant signal handlers can be detected by delivering signals to a process already handling a signal. Similarly, return values of system calls can be changed to cause conditions suitable for attack (e.g., call to drop privileges fails). While STING could be easily extended to perform these attacks, we believe that these cases are more easily handled through static analysis, as standard techniques are available (e.g., lists of non-reentrant system calls, unchecked return value) through tools such as Fortify [28]. For the reasons we have seen, no such standard techniques are available for name resolution attacks.

Solutions. One of the more effective ways we have seen programs defending against improper binding attacks is by dropping privileges. For example, the privileged part of `sshd` drops privileges to the user whenever accessing user files such as `.ssh/authorized_keys`. Thus, even if code is vulnerable to improper binding attacks, the user cannot escalate privileges.

User directory. Administrators running as `root` should take extreme care when in user-owned directories, as there are several opportunities for privilege escalation. For example, we found during our testing that if the Python interpreter is started in a user-owned directory, Python searches for modules in that directory. If a user has malicious libraries, then they will be loaded instead. More race conditions are also exploitable as a user can delete even `root`-owned files in her directory.

Integration with Black-box Testing. We believe that STING can also integrate with other data fuzzing platforms [41]. Such tools need special environments (e.g., attaching to running processes with debuggers) to carry out their tests on running programs. Instead, we can take input from these platforms and use STING to feed such input into running processes. Since STING also takes into account the access control policy, opportunities to supply adversarial data can readily be located.

Deployment. We envision that STING would be deployed during Alpha and Beta testing of distribution re-

leases. We plan to package STING for distributions, so users can easily install it through the distribution's package managers. STING will test various programs as users are running them, and program vulnerabilities found can be fixed before the final release. Being a runtime analysis tool, STING can possibly find more vulnerabilities as it improves its runtime coverage. Even if a small percentage of users install the tool, we expect a significant increase in the runtime coverage, because different users configure and run programs in different ways.

9 Conclusion

In this paper, we introduced STING, a tool that detects name resolution vulnerabilities in programs by dynamically modifying system state. We examine the deficiencies of current static and normal runtime analysis for detecting name resolution vulnerabilities. We classify name resolution attacks into improper binding and improper resource attacks. STING checks for opportunities to carry out these attacks on victim programs based on an adversary model, and if an adversary exists, launches attacks as an adversary would by modifying namespace state visible to the victim. STING later detects if the victim protected itself from the attack, or it was vulnerable. To allow online operation, we propose mechanisms for rolling back the namespace state. We tested STING on Fedora 15 and Ubuntu 11.10 distributions, finding 21 previously unknown vulnerabilities across various programs. We believe STING shall be useful in detecting name resolution vulnerabilities in programs before attackers. We plan to release and package STING for distributions for this purpose.

References

[1] A. Aggarwal and P. Jalote. Monitoring the security health of software systems. In *ISSRE-06*, pages 146–158, 2006.

[2] Aufs. `http://aufs.sourceforge.net/`.

[3] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, Feb. 2011.

[4] G. Banks *et al.* . Snooze: Toward a stateful network protocol fuzzer. In *of Lecture Notes in Computer Science*, 2006.

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

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

[7] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: attacking path explosion in constraint-based test generation. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, 2008.

[8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[10] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

[11] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

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

[13] 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.

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

[15] B. Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[16] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. *Security and Privacy, IEEE Symposium on*, 0:184, 1987.

[17] Container-based checkpoint/restart prototype. `http://lwn.net/Articles/430279/`.

[18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, pages 117–130, 2007.

[19] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[20] 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.

[21] init script x11-common creates directories in insecure manners. `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=661627`.

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

[23] Doup, Adam and Cova, Marco and Vigna, Giovanni. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *DIMVA*, 2010.

[24] W. Drewry and T. Ormandy. Flayer: exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[25] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, 2007.

[26] W. Enck *et al.* Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[27] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[28] Hp fortify static code analyzer (sca). `https://www.fortify.com/products/hpfssc/source-code-analyzer.html`.

[29] P. Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 2007.

[30] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[31] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[32] B. Goyal, S. Sitaraman, and S. Venkatesan. A unified approach to detect binding based race condition attacks. In *International Workshop on Cryptology And Network Security*, 2003.

[33] N. Hardy. The confused deputy. *Operating Systems Review*, 22:36–38, 1988.

[34] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th USENIX Security Symp.*, 2003.

[35] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.

[36] C. Ko and T. Redmond. Noninterference and intrusion detection. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 177–, Washington, DC, USA, 2002. IEEE Computer Society.

[37] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.

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

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

[40] 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.

[41] Peach fuzzing platform. `http://peachfuzzer.com/`.

[42] S. Rueda, D. H. King, and T. Jaeger. Verifying compliance of trusted programs. In *Proceedings of the* 17^{th} *USENIX Security Symposium*, pages 321–334, Aug. 2008.

[43] P. Saxena *et al.* Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[44] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, 2005.

[45] B. Schwarz, H. Chen, D. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[46] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[47] Sharefuzz. `http://sourceforge.net/projects/sharefuzz/`.

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

[49] 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.

[50] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 13:1–13:18, Berkeley, CA, USA, 2008. USENIX Association.

[51] 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.

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

[53] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *ACSAC*, 2000.

[54] H. Vijayakumar, G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger. Integrity Walls: Finding Attack Surfaces from Mandatory Access Control Policies. In *AsiaCCS*, 2012.

[55] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, 2010.

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

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

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

[59] C. P. Wright and E. Zadok. Unionfs: Bringing File Systems Together. *Linux Journal*, pages 24–29, December 2004.

[60] W. Xu, E. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *In 15th USENIX Security Symposium*, pages 121–136, 2006.