A Rose by Any Other Name or an Insane Root?[*] Adventures in Name Resolution

Hayawardh Vijayakumar, Joshua Schiffman, and Trent Jaeger
Systems and Internet Infrastructure Security Lab
Penn State University
Email: {hvijay,jschiffm,tjaeger}@cse.psu.edu

Abstract

Namespaces are fundamental to computing systems. Each namespace maps the names that clients use to retrieve resources to the actual resources themselves. However, the indirection that namespaces provide introduces avenues of attack through the *name resolution process*. Adversaries can trick programs into accessing unintended resources by changing the binding between names and resources and by using names whose target resources are ambiguous. In this paper, we explore whether a unified system approach may be found to prevent many name resolution attacks. For this, we examine attacks on various namespaces and use these to derive invariants to defend against these attacks. Four prior techniques are identified that enforce aspects of name resolution, so we explore how these techniques address the proposed invariants. We find that each of these techniques are incomplete in themselves, but a combination could provide effective enforcement of the invariants. We implement a prototype system that can implement these techniques for the Linux filesystem namespace, and show that invariant rules specific to each, individual program system call can be enforced with a small overhead (less than 3%), indicating that fine-grained name resolution enforcement may be practical.

1 Introduction

Namespaces are fundamental to computing systems. Namespaces map the names that clients use for retrieving and sharing resources to the actual resources. For example, a file's pathname is linked to a unique file inode that represents the physical resources that file pathname addresses. The process of translating the name to the resource in a namespace is called *name resolution*. Filesystem, process, and network namespaces are well-known examples, but component-specific namespaces like system services (e.g., D-Bus methods), middleware (Android), and web infrastructure (e.g., URLs) exist. Namespace resolution is a key issue in any distributed system design [6, 20], so we expect new namespaces will continue to emerge as new software architectures are developed.

However, the indirection that namespaces provide introduces avenues of attack through the name resolution process. By altering the bindings between names and the intended resources, adversaries can trick victim processes into accessing untrustworthy resources. For example, time-of-check-to-time-of-use attacks [2] (TOCTTOU) enable an adversary to change the file inode bound to a pathname after the victim has checked to determine the user's access rights to that file. Other attacks rely on the ambiguity surrounding which actual resource may be associated with a name. A confused deputy problem [15] may be created when an adversary provides a file path to a privileged server that fails to recognize the path is a link to a privileged inode, like a password file.

The typical approach to defend against name resolution attacks is to provide programmers with API changes that enable them to detect and prevent such attacks themselves. For example, system calls have been extended with flags that prevent a process from creating a file if one already exists with that name (prevents squatting) or following a symbolic link (limit the accessible files). However, programmers continue to make errors because some programs require such risky functions and programmers simply fail to add the necessary checks. Also, researchers have found that it is difficult to prevent such attacks in general. Hu and Dean proposed a system solution to prevent the access TOCTTOU vulnerability [11], later extended by Tsafir *et al.* [24], but others found both solutions flawed [4], leading to the requirement that any comprehensive defense against race conditions requires an accurate model of the programs that it protects. However, this limitation should not prevent us from exploring defenses, but rather we must build defenses understanding these limitations.

In this paper, we explore options for a unified system solution to prevent name resolution attacks. We cite the

[*] Apologies to William Shakespeare and his fans.

experience with preventing memory errors as a motivation. The prevailing thought ten years ago was that programmers could, with the right training and tools, prevent such errors, but this proved to be ineffective. Instead, incomplete defenses were proposed that do not require programmer input [1], and those with sufficiently low overhead, such as canaries [10] and non-executable memory [13], have now been adopted to prevent many types of buffer overflow vulnerabilities.

The question we explore is whether a system approach based on a combination of low-cost, incomplete defenses that do not require programmer input may be found to prevent many name resolution attacks. To find the answer to this question, we identify the requirements for name resolution attacks and explore the effectiveness of candidate defenses. A key issue is that name resolution attacks involve two components, one that manages the namespace and one that uses the namespace, and it is necessary for the two to collaborate to prevent such attacks.

The paper continues with Section 2 that outlines the security model for name resolution and describe a variety of attacks against namespaces. Section 3 develops name resolution invariants and evaluates the effectiveness of previous defenses using these invariants. In Section 4, we examine how such mechanisms may be composed into a coherent defense mechanism, using the filesystem namespace as the example. Finally, Section 5 explores how this approach may be generalized to apply to any name resolution mechanism and the challenges in implementing the mechanism for particular namespaces.

2 Background

In this section, we describe attacks on name resolution and define our security model.

2.1 Name Resolution

Name resolution is performed by a name server. When a client provides a name, the server returns a resource reference. Each resolution is specific to a namespace, which may map names to resources one-to-one, many-to-one (e.g., many file pathnames may refer to one file inode), or one-to-many (e.g., one name may refer to many possible resources of which one is selected usually). At present, we have not seen a many-to-many namespace. Name servers may also enforce access control over the client's ability to use the associated resource. Once the resource reference is returned the client may use that resource within the scope of her access rights.

We briefly examine two different name resolutions to demonstrate why security problems may occur in name resolution. First, virtual address resolution converts a virtual address (the name) to a physical address (the resource). In modern systems, page table architectures may contain one or more levels of mappings, where the result of a mapping is the availability of the associated physical frame backing the physical address. Virtual address resolution does not suffer from vulnerabilities because it is quite restrictive: the mapping is one-to-one, defined by a trusted server, and only modifiable by the client itself or the server.

Second, the file pathname resolution process is similar, but presents some additional flexibility to allow multiple parties to manage the namespace. An important issue is that multiple clients may bind names to file inodes, where some (the file owners) can choose the files' access rights as well. Clients can create resources in locations dictated by their file system permissions, but some locations are shared (e.g., /tmp). Also, clients may create multiple names for the same file inode, using links. Naming a file inode does not require access to the file, as a symbolic link is simply a path. As a result, file resolution may depend on bindings between file names and inodes defined by untrusted clients. Also, the mapping between file names and inodes may be changed by clients at any time. This flexibility enables applications to create, manage, and share their files easily, but also provides opportunities for adversaries.

2.2 Security Model

We now identify our security model for reasoning about vulnerabilities described in this section. First, *name servers* are trusted by all processes to perform name resolution using their built-in algorithm, based on the current state of the namespace. The adversary does not control the name server. Second, *victim processes* are the targets of potential attack. Any process could be a potential victim, but they are assumed not to be under the control of an adversary. In particular, we assume that they generate name resolution requests that are consistent with the programmers' model of the expected namespace. Finally, *adversaries* may control any external host or unprivileged processes on the victim's host. We assume that adversaries may use any permission available to the hosts and processes they control.

Threats emerge from the distinction between the programmers' model of the expected name resolution and the actual resolution. Vulnerabilities are then possible if the adversary is able to redirect the victim to a resource with different security semantics than the expected resource. For example, if an adversary can get a victim to obtain a reference to a file whose content is under the adversary's control (i.e., is low integrity) when a trusted file is expected, then the victim can be compromised. Alternatively, an adversary may also provide a name to a high-integrity resource that the adversary cannot modify directly and trick the victim, who can modify this resource, to perform modifications dictated by the adversary [15].

2.3 Example Attacks

There are a variety of name resolution attacks, and we examine some specific instances that explore various facets of namespace problems. We use these attacks to identify four categories of problems with namespaces.

Namespace Pre-binding Attack. The namespace pre-binding attack allows adversary processes to create untrusted bindings that help them masquerade as another entity.

Example 1 - System V shared memory. System V shared memory uses a "key" namespace, as memory pages are not addressible by the filesystem namespace. The key resolves to a shared memory offering, and both the offering and the sharing processes need to know their unique key. Since this key namespace is shared, the same key can be offered by processes running under subjects with different integrity. Thus, if an adversary process can associate a shared memory region with a key before a trusted process process can, it can masquerade as that trusted process. A generalization of this problem is IPC squat [3].

Example 2 - XenStore. XenStore provides a store of key-value pairs that contain information about running virtual machines under the Xen VMM. Parts of this namespace contain critical information, and must be writable only by the privileged domain, dom0. However, there have been at least two cases where, due to improper permissions on the namespace, VMs were able to write arbitrary values onto critical keys, thus introducing low-integrity bindings. dom0 accessed and used this information without realizing it was a bad binding.

Example 3 - Linux filesystems. A common attack on Linux filesystems is link following. In this attack, a victim process uses some temporary files with predictable names. It however, does not check that these temporary files are already bound. Thus, an adversary process simply creates a symbolic link with the predictable name to any critical file (e.g., `/etc/passwd`). The victim program corrupts the file, without realizing that it followed a bad binding.

Namespace Rebinding Attack. In this type of attack, an adversary rebinds a trusted binding to an unintended resource. The ability to rebind makes an adversary more powerful than the ability to just bind.

Example 4 - Linux filesystems. A common attack on Linux filesystems is TOCTTOU. Here, a victim process checks to see if the resource it accesses is valid, and then uses it. However, in the time between the check and use, an adversary could update the mapping to make it point to a file of her choice.

Example 5 - HTML DOM. There are a few DOM name resolution functions in JavaScript. In particular, the `getElementById` function should "access the first element with the specified id" [25]. Although we do not know of such exploits, an adversary who is able to inject just plain HTML into a page (e.g., many forums allow this), could re-bind the resolution of an already existing HTML element to the adversary's injected element by giving it the same ID, provided it occurs earlier on the page. JavaScript and other code that depends on this element would now resolve to the adversary-controlled element.

Namespace Multi-binding Attack. The previous namespaces mentioned offer either one-to-one or many-to-one mappings. However, some namespaces use one-to-many bindings. The resolution algorithm then chooses one among those bound to the name. An adversary can create a binding and hope that the resolution algorithm chooses it over other, legitimate choices.

Example 6 - Android IPC. Android enables processes to communicate via the Binder IPC mechanism. We note that Android IPC is vulnerable to IPC squat [3]. We focus on a different problem here. A process can request to connect to a foreground Activity, or a background Service, by sending an Intent message, that identifies the recipient of the request. Intents can explicitly identify a recipient, or implicitly, by allowing the OS to select among a possible set of processes that have registered to handle that Intent. If an implicit Intent can be handled by multiple foreground activities, the user is given a choice to select one. However, if an implicit Intent can be handled by multiple background services, any one is chosen at random [8]! Thus, an untrusted activity or service registering on an Intent allows unexpected resolutions.

Example 7 - D-Bus. Another example of a one-to-many resolution algorithm is in D-Bus. D-Bus is used by processes for IPC. Servers group multiple related methods into an interface. If a client sends a method without specifying an interface, the bus daemon will try to locate any interface with a matching method signature.

Improper Name Attack. The previous classes of attacks are due to unexpected name resolution. However, programs may supply an unintended name for resolution. This situation occurs most commonly due to bugs in programs, where an adversary controls the supplied name in ways she should not.

Example 8 - Directory Traversal. As an example, consider directory traversal attacks in webservers. The adversary supplies `../../etc/passwd`, and instead of fetching a HTML file, the webserver serves the password file. In this case also, an unexpected resource is fetched, though it is the requesting program that is at fault, and not the name resolution. We contend that the namespace is the proper place to defend against such attacks, and the reso-

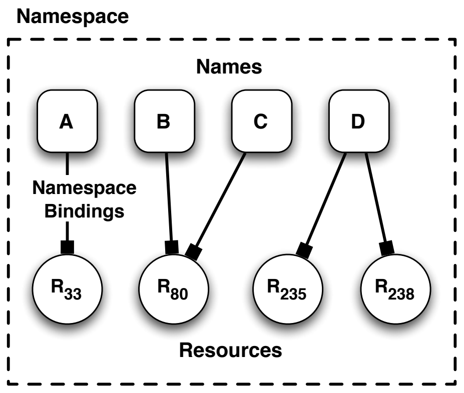


Figure 1: A namespace has names that are resolved to resources through bindings.

lution algorithm can simply fail if it knows the resource is not a proper one for a particular program context.

Finally, we note that traditional access control of processes and resources (e.g., files) in operating systems cannot completely solve the problem of unexpected resolution in the filesystem namespace, as a single program might legitimately need access to adversary-controlled resources through certain interfaces. However, some resolutions are unacceptable in certain situations. We need to protect against unacceptable resolutions (i.e., the namespace resolving to an adversary-controlled resource when a high-integrity resource is expected, and vice-versa). Thus, we need to consider a method that reasons about the context in which requests are made to namespaces to restrict resolutions properly.

3 Name Resolution Invariants

Based on the attacks above, we define name resolution invariants for specifying the requirements for secure name resolution. Figure 1 shows a conceptual model for mapping names to resources, where each name may be bound to one or more resources and a resource may have multiple possible names. Using this model, we examine how four different types of defenses may be used to prevent name resolution attacks, finding that they each provide partial solutions.

System Resource Restriction. The first defense restricts the set of resources that may be retrieved from a name resolution operation. For example, a network firewall [7, 17] restricts which IP addresses may be accessed by a particular system interface (e.g., a port), regardless of the result from a DNS resolution. The aim is to block resolutions that are known to be insecure, such as redirecting mail requests to an untrusted server. Thus, even though the DNS binding between a name and an IP address may be untrusted, the firewall protects the system from being redirected. However, if an IP address may lead to a malicious resource, the firewall may still allow it, should the system's function require such operations (e.g., web client access). Also, firewalls require policy to be specified, which is a manual process. We highlight firewalls rather than access control mechanisms, in general, because firewalls can restrict access per system interface. Access control instead gives the process' full rights to each program system call. We discuss the importance of this in the next section.

Capabilities. A second defense is to circumvent the need for name resolution by providing access to the necessary resources directly. This is the aim of pure capability systems [14, 23], which restrict all naming to be performed using capabilities. That is, each process can only access a resource for which it has a capability. Each process has a capability to a memory region containing a set of capabilities, and other capabilities may be obtained from other processes indirectly. However, confinement is a problem for capability systems [16] because we may want to prevent a process from being able to obtain a capability for an unauthorized object, even though it can communicate with another process with that capability. While EROS can prevent a process from obtaining capabilities that would violate the *-property [23], it cannot confine processes arbitrarily. SCAP [16] and ICAP [12] proposed mechanisms to authorize capability use and propagation, respectively, but writing such policies will often depend on naming objects. The Capsicum system takes this approach to an extreme, obtaining capabilities when processes start [26]. However, one still has to check name resolution when those capabilities are collected. In general, developers find namespaces convenient, so replacing them entirely with pure capability systems seems quite unlikely. Nonetheless, where names can be converted to capabilities securely, the use of capabilities prevents ambiguity due to name rebinding.

Namespace Management. A third defense restricts the ability to modify namespace mappings used in resolution. It has long been recommended that processes have a private namespace [18, 20], and Linux has supported per-process namespaces since version 2.4.19. A per-process namespace enables a process to "unshare" resources by creating their own namespace mappings for a resource. This is employed by container-based virtualization efforts, such as OpenVZ [19], which aim to provide disjoint namespaces for processes in their containers. Also, D-Bus employs per-session namespaces, which are specific to each logged-in session. Per-process namespaces are typically used for convenience (e.g., to run software that depends on different versions of a file typically in the same path), but there is also a potential for reducing name resolution attacks. For example, a process may protect itself from compromise by only using namespace bindings it or a trusted process defined. Unfortunately, this may not always be practical because servers may need to retrieve files defined by low-integrity subjects in their namespaces.

Chari et. al [5] aim to ensure that only high-integrity bindings are resolved in the Linux filesystem shared namespace, by enforcing that only high-integrity subjects (`root` and the requesting user) have permission to modify any component of a pathname that is being resolved (by checking directory permissions). For cases where this does not hold, they use additional heuristics on the bindings to determine whether the resource resolved is appropriate. By viewing the problem more broadly, we envision that we can control name resolution directly using resource constraints (e.g., is the resource appropriate for the request?) and using per-process namespaces to prevent unexpected rebindings.

Program Resource Restrictions. Finally, a fourth defense is based on knowledge of the program intent. Cai *et al.* argue that any kernel (i.e., name server) race detector must have side information about the program or it otherwise is limited in its function [4] (i.e., either has false positives or false negatives). As capability and namespace management solutions eliminate races by preventing changes in bindings, they are not susceptible to these limitations (although they have other limitations). However, the firewall defense above and other custom race detectors that would run in the name server [11, 24] suffer some limitation. The challenge is how to get this side information. Programmers may specify such information, but they are typically loathe to annotate their programs and often make errors. The alternative is to perform program analyses to identify races and the scope of resources required for each system call. However, runtime analysis may be incomplete and static analysis may be imprecise.

Based on this analysis of known defenses, we identify four distinct ways to prevent name resolution attacks, but each has limitations. Our goal is to define invariants for name resolution, and then explore how we can combine such techniques to greatly limit an adversary's ability to compromise victims, eliminating attacks entirely in some cases.

We thus propose the following *name resolution invariants*:

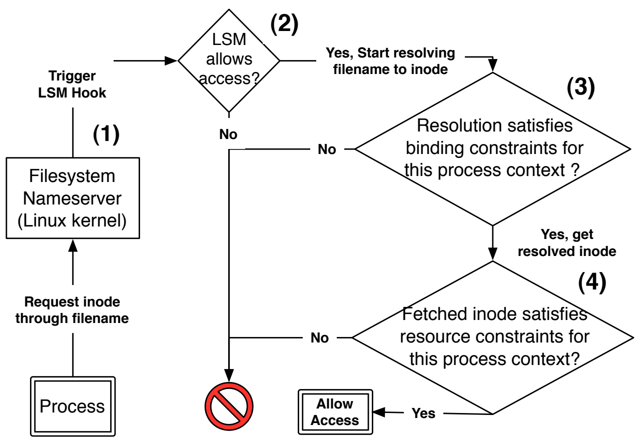
i-resource*: Each name resolution $R = (s, n, p)$ in* namespace *s of* name *n by* process context *p must restrict the resource output from a resolution of s to* authorized resources *for p and n.*

i-binding*: Each name resolution $R = (s, n, p)$ in* namespace *s of* name *n by* process context *p must restrict the namespace bindings used in resolutions of s to those defined by* authorized subjects *for p and n.*

We note that each name resolution should enforce its requirements of the invariants *i-binding* and *i-resource* – only one or both may apply. To see why this is the case, consider the following two cases for Linux filesystems. First, consider the directory traversal attack in Section 2.3. In this case, even if the webserver follows only trusted bindings, it will end up with a resource that is outside the expected scope. Thus, *i-binding* is irrelevant here, but *i-resource* needs to be enforced. On the other hand, consider a program accessing `/tmp`. It is known that adversaries can write to `/tmp`, and there is no use placing restrictions such as that the resource being fetched should be in `/tmp` (*i-resource* is irrelevant). What is needed is a guarantee that the particular file being resolved is through trusted bindings (*i-binding*).

We define invariants on *process context* rather than simply processes in general, because the *i-binding* and *i-resource* constraints on each process execution context that makes the name resolution request may be different. A process context is defined by the state of the process at the time of a system call (e.g., process's execution call stack). It is well-known that many attacks are caused because a process expects a resource with particular properties at a particular system call, but the adversary may use name resolution ambiguity to produce a resource with different (i.e., adversary-controlled) properties. For example, a text editor might accept adversary-controlled low-integrity files to edit, but only high-integrity files as shared libraries – different constraints apply in different process contexts. Hence, the name resolution invariants categorize the resource properties into two types: *i-binding* for those describing the binding used to retrieve the resource and *i-resource* for those properties of the retrieved resource itself. Process context allows for different properties per execution context (e.g., call stack) of the process.

Given these invariants, we find that the prior defenses relate to the invariants in the following way. System resource restrictions enforce the *i-resource* invariant. Capabilities enforce a special case of the *i-resource* invariant: ensuring that the same resource is used in multiple name resolutions. Namespace management enforces the *i-binding* invariant. Finally, program resource restrictions may imply *i-binding* and *i-resource* restrictions, but these restrictions may be incomplete. That is, the quality of program resource restrictions depends on the accuracy of program analyses (static or dynamic) and the effort that programmers make in helping to improve the results of such analyses. Since we have little control of what the programmers may or may not do, the other mechanisms are necessary to provide defense-in-depth to prevent name resolution attacks.



(2)
LSM allows access?
Trigger LSM Hook
Yes, Start resolving filename to inode
(3)
(1)
No
No
Resolution satisfies binding constraints for this process context ?
Filesystem Nameserver (Linux kernel)
Yes, get resolved inode
(4)
Request inode through filename
No
Fetched inode satisfies resource constraints for this process context?
Process
Allow Access
Yes

Figure 2: Our prototype for the Linux filesystem enforces invariant *i-binding* at (3) and *i-resource* at (4).

4 Experiment

Motivated by previous defenses against memory attacks, we examine the efficacy of building a system mechanism to enforce the name resolution invariants. In general, a system mechanism should implement the four defenses from the previous section efficiently, and require little or no programmer effort to use to prevent many attacks. We explore such defenses in the context of the Linux filesystem, but we envision that such defenses can be applied to other namespaces similarly.

Experimental Platform. For our experiment, we built a prototype to enforce the name resolution invariants for the Linux filesystem. Our in-kernel mechanism mediates all filesystem name resolutions. To do this, we extended the SELinux security module to compare a file inode and its security labeling against the expected inode based on the requested name and the process' state during the request. We also evaluated the performance of our proof of concept mechanism in enforcing the name resolution invariants per individual process system calls (see below). Another issue we explore is how easy it is to identify the name resolution invariant constraints for each process context. We explore where the policies may be obtained without manual input, particularly in assessing program resource restrictions.

Prototype Implementation. We implemented our filesystem namespace enforcement prototype in the Linux 2.6.35 kernel as shown in Figure 2. When a process makes a filesystem related system call, (1) the kernel's filesystem name resolver converts the path to an inode. The inode is then (2) passed through the normal OS access control mechanisms (DAC and LSM access control checks). If the access is valid, during the resolution process, we (3) check if the binding used to resolve the inode satisfies any constraints for that particular process context (if any exist), thus maintaining *i-binding*. Next, (4) we verify that the fetched inode satisfies any constraints required for that particular process context (if any exist), maintaining invariant *i-resource*. We currently approximate *i-binding* by asserting that some resolutions use a private namespace, approximately the basic mechanism of Chari *et al.* [5]. Only resolutions that pass both the *i-resource* and *i-binding* rules are accepted. Currently, we use the program counter of the calling process as the process context, to enable applying different *i-resource* and *i-binding* requirements at each point where the program requests a name resolution.

Security Results. We now examine how our prototype integrates into the four prior defenses.

System Resource Restriction. Our prototype can verify that the resource retrieved as a result of a name lookup is within a set of *authorized resources* for that particular access to satisfy invariant *i-resource*. The authorized resources are represented by a set of inodes and a set of security labels. The retrieved inode must be a descendant of one of the inodes in the authorized resources and have a security label in the set of authorized security labels. For example, to prevent link and parsing vulnerabilities, the inode must be a descendant of the authorized inode. To prevent squatting and untrusted input, the label of the inode must be in the authorized set, which are determined by application. Use of untrusted, dynamic libraries can be prevented by limiting the inodes to specific subtree (e.g., in `/lib` and `/usr/lib`) or to specific labels (e.g., SELinux label `lib_t`).

Capabilities. Capabilities are a direct reference to a resource, and elide the need for name resolution. We can mimic capabilities in our prototype by requiring use of a particular resource from a prior resolution. For example, to prevent TOCTTOU attacks, the prototype can enforce that the vulnerable system call (e.g., `open()` after `access()`) use a previously resolved inode (in this case, the same inode as was referenced in the `access()` call). In effect, we have assigned a capability to the `open()` interface. Determining when to require the same resource is a challenge. Researchers have identified the combinations of system calls that are vulnerable to TOCTTOU attacks [22] to guide the identification of such requirements from program intent. We use such information to collect likely TOCTTOU cases from program resource restrictions, described further below.

Namespace Management. As mentioned above, we implemented a mechanism to restrict some process contexts to use the existence of a per-process namespace for resolution, emulating the basic enforcement mechanism in Chari *et al.* [5]. This can prevent a number of attacks, such as ones against squatting on temporary files. In this way, only a process can only resolve names to the resources that it created in /tmp, and adversary processes cannot modify the bindings in a victim's /tmp directory, as they cannot change this namespace. As Chari *et al.* noted [5], there may be legitimate cases where untrusted bindings may be used, and they implement some further defenses based on heuristics for these more complex cases. We have not yet implemented a general mechanism for enforcing more complex rules, although those authors have shown that the necessary binding information is accessible. Also, recall that we argued previously that using *i-resource* invariant constraints instead may be more effective than trying to control the binding alone.

Program Resource Restrictions. By knowing more about what a program is expected to do, we can further restrict the scope of resources accessible at various process contexts. We explore the effectiveness of runtime program analysis to generate name resolution constraints for process contexts. MAC policy writers use runtime analysis of programs to determine their least privilege policies [21]. Many Linux packages now include a test suite for evaluating the function and robustness of these programs. We run these test suites to gather runtime information. In our experiment, we use the runtime analysis to identify likely TOCTTOU system calls to generate constraints (i.e., cases where the same name is used across known vulnerable system call sequences). We also believe that runtime analysis will be useful for identifying process contexts that should be limited to trusted resources (e.g., to prevent search path vulnerabilities and squatting attacks). We envision that a combination of both static and runtime analyses should be performed, although a key problem is that the actual invariant constraints may depend on the particular configuration of the program.

Prototype Performance. A performance concern is that we might have a large rule base specifying valid resolutions for different running process contexts, and thus traversing rules might incur much overhead. Thus, we ran the system on a simple rule base to assess the basic performance. Our rule base consists of rules that protect pairs of system calls against TOCTTOU attacks for various processes, by making sure the latter system call is given capabilities to access only the object that the former system call has checked for [22] (enforcing *i-resource*).

For a macrobenchmark, we chose compiling Apache, as it performs a relatively high number of filesystem operations, and would represent close to a worst case for us. We found an overhead of 2.8%. We also measured the overhead on individual system calls that access the filesystem namespace, They ranged from 4% for read/write calls, 17% for open/close calls, to 27% for stat/access calls. The access system call performs very little apart from checking for access, and hence has a maximum overhead. However, this is amortized over the higher cost for other system calls, as we find in the macrobenchmark.

5 Discussion and Future Work

Our experiment shows that it may be possible to control name resolution by leveraging a combination of the four prior defenses to enforce name resolution invariants per process context. However, this preliminary investigation raises several, interesting research questions:

- *Can we enforce both i-binding and i-resource invariants using a single mechanism?* Current approaches either enforce bindings or resources retrieval, but not both. A coherent approach could utilize whichever approach is most effective for a particular defense.
- *Can the same name resolution enforcement techniques be applied to all namespaces?* We see that some namespaces, such as Android and DOM objects, use namespaces where multiple objects may have the same name. It would seem that *i-binding* or *i-resource* could enforce such resolutions, but are there application-specific requirements that may not easily be captured in either of these invariants?
- *Can we generate the most efficient invariants for enforcement?* Then, a question becomes whether the most efficient enforcement policy can be determined from the programs. For example, to prevent TOCTTOU attacks, we may simply restrict vulnerable operations to use the same resource, rather than enforcing bindings.
- *Does it really matter whether the runtime analysis generates complete name resolution policies?* Using runtime analysis, we can collect name resolution policies, but these may be too conservative. For example, there may be authorized cases that are not seen by the runtime analysis. Therefore, there may be acceptable cases that need to be added to the policy after the fact. Rather than extending policies manually, we can examine techniques for generating policies automatically, such as Bouncer [9].
- *Can we develop a general approach to name resolution enforcement?* Given an enforcement mechanism, automated policy generation, and automated policy optimization, can we build a library that builds and maintains name resolution enforcement policies automatically for all kinds of namespaces?

If such questions can be answered in the affirmative, then attacks based on name resolution can be greatly reduced, as buffer overflow attacks have been. However, as Cai *et al.* showed [4], developing guaranteed defenses against TOCTTOU attacks (i.e., one form of name resolution attacks) depends on an accurate understanding of program semantics, which may be beyond what is computationally tractable. Thus, name resolution enforcement provides defense-in-depth to prevent attacks that may be overlooked by programmers.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of CCS '05*. ACM, 2005.

[2] M. Bishop and M. Dilger. Checking for Race Conditions in File Accesses. *Computing systems*, 1996.

[3] J. Burns. Developing Secure Mobile Applications For Android.

[4] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[5] S. Chari, S. Halevi, and W. Venema. Where do you want to go today? Escalating Privileges by Pathname Manipulation. In *NDSS*, 2010.

[6] D. R. Cheriton. The v Distributed System. *Communications of the ACM*, 1988.

[7] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, 2003.

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

[9] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of SOSP '07*. ACM, 2007.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

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

[12] L. Gong. A Secure Identity-Based Capability System. In *Proceedings of IEEE Symposium on Security and Privacy*, 1989.

[13] Homepage of PaX. `http://pax.grsecurity.net/`, 2008.

[14] N. Hardy. The KeyKOS architecture. *Operating Systems Review*, 19(4):8–25, Oct. 1985.

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

[16] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Security and Privacy, IEEE Symposium on*, 1984.

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

[18] R. Needham. "Names". *Distributed systems, S. Mullender, ed.,*, 1989.

[19] OpenVZ. `http://wiki.openvz.org/Main_Page`.

[20] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *UKUUG Proc. of the Summer 1990 Conf*, 2006.

[21] N. Provos. Improving host security with system call policies. In *Proceedings of USENIX SS '03*, 2003.

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

[23] J. S. Shapiro and S. Weber. Verifying the EROS Confinement Mechanism. In *IEEE Symposium on Security and Privacy*, 2000.

[24] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of the 6th USENIX FAST*. USENIX, 2008.

[25] w3schools. HTML DOM Document getElementById() Method. `http://www.w3schools.com/jsref/met_doc_getelementbyid.asp`.

[26] R. Watson, J. Anderson, and B. Laurie. Capsicum: practical capabilities for UNIX. In *Proceedings of USENIX SS '10*, 2010.