Towards Adaptive Process Confinement Mechanisms

COMP5900I Literature Review

William Findlay

October 30, 2020

Abstract

[Come back hither when done.]

1 Introduction

Restricting unprivileged access to system resources has been a key focus of operating systems security research since the inception of the earliest timesharing computers in the late 1960s and early 1970s [11, 23, 34]. In its earliest and simplest form, access control in operating systems meant preventing one user from interfering with or reading the data of another user. The natural choice for many of these early multi-user systems, such as Unix [34], was to build access control solutions centred around the user model—a design choice which has persisted in modern Unix-like operating systems such as Linux, OpenBSD, FreeBSD, and MacOS. Unfortunately, while user-centric permissions offer at least some protection from other users, they fail entirely to protect users from *themselves* or from their own *processes*. It was long ago recognized that finer granularity of protection is required to truly restrict a process to its desired functionality [25]. This is often referred to as *the process confinement problem* or *the sandboxing problem*.

Despite decades of work since Lampson's first proposal of the process confinement problem in 1973 [25], it remains largely unsolved to date [15]. This begs the question as to whether our current techniques for process confinement are simply inadequate for dealing with an evolving technical and adversarial landscape. In this literature review, I argue that, in order to solve the process confinement problem, we need to rethink the status quo in process confinement, and instead move towards *adaptive* process confinement mechanisms.

1.1 Defining Adaptive Process Confinement

Here, I define adaptive process confinement mechanisms as those which greatly help defenders confine their processes, are easily adoptable across a variety of system configurations, and are robust in the presence of attacker innovation. Roughly, this definition can be broken down into the following properties:

P1. Robustness to attacker innovation. An adaptive process confinement mechanism should continue to protect the host system, even in the presence of attacker innovation. That is, it should be resistant to an adaptive adversary.

P2. Adoptability. An adaptive process confinement mechanism should require minimal effort to adopt on a variety of system configurations. It should work out of the box on the majority of target systems and should be deployable in a production environment without major security or stability concerns.

P3. Reconfigurability. An adaptive process confinement mechanism should be highly reconfigurable based on the needs of the end user and the environment in which it is running. This reconfiguration could either be automated, semi-automated, or manual, but should not impose significant adoptability barriers (c.f. P2).

P4. Transparency. An adaptive process confinement mechanism should be as transparent to the end user as possible. It should not get in the way of ordinary system functionality, and should not require the modification of application source code in order to function. Further, its base functionality should not require significant user intervention, if at all.

P5. Usability (by non-experts). An adaptive process confinement mechanism should maximize its usability such that it is usable by largest and most diverse set of defenders possible. In particular, it should not require significant computer security expertise from its users.

1.2 Outline of the Literature Review

To argue the case for adaptive process confinement, we need to understand the existing process confinement literature from an adaptive perspective. To that end, I present a novel taxonomy of the existing literature, categorizing process confinement mechanisms as either *maladaptive* or *semi-adaptive* using the adaptive properties (items P1–P5) outlined above. In light of this categorization, I then discuss what the move toward *truly adaptive* process confinement mechanisms might look like.

The rest of this paper proceeds as follows. Section 2 presents the process confinement threat model. Section 3 examines the status quo in process confinement, which is predominantly made up or maladaptive and semi-adaptive solutions. Section 5 discusses moving toward truly adaptive process confinement mechanisms and argues that new operating system technologies coupled with well-known techniques from the anomaly detection literature may enable a paradigm shift in this direction. Section 6 concludes.

2 The Process Confinement Threat Model

To understand why process confinement is a desirable goal in operating system security, we must first identify the credible threats that process confinement addresses. To that end, I first describe three attack vectors (items A1 to A3), followed by three attack goals (items G1 to G3) which highlight just a few of the threats posed by unconfined processes to system security, stability, and user privacy.

A1. Compromised processes. Unconfined running processes have classically presented a valuable target for attacker exploitation. With the advent of the Internet, web-facing processes which handle untrusted user input are especially vulnerable, particularly as they often run with heightened privileges [10]. An attacker may send specially crafted input to the target application, hoping to subvert its control flow integrity via a classic buffer overflow, return-oriented programming [36], or some other means. The venerable

Morris Worm, regarded as the first computer worm on the Internet, exploited a classic buffer overflow vulnerability in the `fingerd` service for Unix, as well as a development backdoor left in the `sendmail` daemon [41]. In both cases, proper process confinement would have eliminated the threat entirely by preventing the compromised programs from impacting the rest of the system.

A2. Semi-honest software. Here, I define semi-honest software as that which appears to perform its desired functionality, but which additionally may perform some set of unwanted actions without the user's knowledge. Without putting a proper, external confinement mechanism in place to restrict the behaviour of such an application, it may continue to perform the undesired actions ad infinitum, so long as it remains installed on the host. As a topical example, an `strace` of the popular Discord [16] voice communication client on Linux reveals that it repeatedly scans the process tree and reports a list of *all applications* running on the system, even when the "display active game" feature[1] is turned off. This represents a clear violation of the user's privacy expectations.

A3. Malicious software. In contrast to semi-honest software, malicious software is that which is expressly designed and distributed with malicious intent. Typically, this software would be downloaded by an unsuspecting user either through social engineering (e.g. fake antivirus scams) or without the user's knowledge (e.g. a drive-by download attack). It would be useful to provide the user with a means of running such potentially untrustworthy applications in a sandbox so that they cannot damage the rest of the system.

G1. Installation of backdoors/rootkits. Potentially the most dangerous attack goal in the exploitation of unconfined processes is the establishment of a backdoor on the target system. A backdoor needn't be sophisticated—for example, installing the attacker's RSA public key in `ssh`'s list of authorized keys would be sufficient—however the most sophisticated backdoors may result in permanent and virtually undetectable escalation of privilege. For instance, a sophisticated attacker with sufficient privileges may load a *rootkit* [6] into the operating system kernel, at which point she has free reign over the system in perpetuity (unless the rootkit is somehow removed or the operating system is reinstalled).

G2. Information leakage. An obvious goal for attacks on unconfined processes (and indeed the focus of the earliest literature on process confinement [25]) is information leakage. An adversary may attempt to gain access personal information or other sensitive data such as private keys, password hashes, or bank credentials. Depending on the type of information,

[1] This feature allows Discord to report, in the user's status message, what game the user is currently playing. This appears to be the original motivation behind scanning the process tree.

an unauthorized party may not even necessarily require elevated privileges to access it—for instance, no special privileges are required to leak the list of processes running on a Linux system, as in the case of Discord [16] highlighted above.

G3. Denial of service. A compromised process could be used to mount a denial of service attack against the host system. For example, an attacker could take down network interfaces, consume system resources, kill important processes, or cause the system to shut down or reboot at an inopportune moment.

As shown in the examples above, unconfined processes can pose significant threats to system security and stability as well as user privacy. With the advent of the Internet, many of these threats are now exacerbated. Unconfined network-facing daemons continually process untrusted user input, resulting in an easy and potentially valuable target for attacker exploitation. Email and web browsers have enabled powerful social engineering and drive-by download attacks which often result in the installation of malicious software. Semi-honest software can violate user expectations of security and privacy by performing unwanted actions without the user's knowledge. It is clear that a solution is needed to mitigate these threats—for this, we turn to process confinement. Unfortunately, process confinement is not yet a solved problem [15], and so the exploration of new solutions is necessary.

3 The Status Quo in Process Confinement

[TODO: new intro]

Criteria for Selecting Process Confinement Mechanisms [TODO]

3.1 Low Level Techniques

In this section, I discuss many of the low level techniques used to implement process confinement. Notably, many of these techniques were not designed expressly for the purpose of directly confining processes. Rather, they are often used in *combination* by higher level process confinement mechanisms, many of which are discussed in Section 3.2. Despite the fact that many of these techniques come pre-enabled on their respective operating systems, they all suffer to some in extent in the robustness, reconfigurability, transparency, and usability categories, which cements their position as maladaptive approaches.

Discretionary Access Control Discretionary access control (DAC) forms the most basic access control mechanism in many operating systems, including popular commodity operating systems such as Linux, MacOS, and Windows. First formalized in the 1983 Department of Defense standard [44], a discretionary access control system partitions system objects (e.g. files) by their respective owners, and allows resource owners to grant access to other users at their discretion. Typically,

systems implementing discretionary access control also provide a special user or role with the power to override discretionary access controls, such as the superuser (i.e. `root`) in Unix-like operating systems and the Administrator role in Windows.

While discretionary access controls themselves are insufficient to implement proper process confinement, they do form the basis for the bare minimum level of protection available on many operating systems, and are therefore an important part of the process confinement discussion. In many cases, user-centric discretionary access controls are abused to create per-application "users" and "groups". For instance, a common pattern in Unix-like systems such as Linux, MacOS, FreeBSD, and OpenBSD is to have specific users reserved for security-sensitive applications such as network-facing daemons. The Android mobile operating system takes this one step further, instead assigning an application- or developer-specific UID (user ID) and GID (group ID) to *each* application installed on the device [22].

In theory, these abuses of the DAC model would help mitigate the potential damage that a compromised application can do to the resources that belong to other users and applications on the system. However, due to the discretionary nature of DAC, there is nothing preventing a given user from simply granting permissions to all other users on the system. Further, the inclusion of non-human users into a user-centric permission model may result in disparity between an end-user's expectations and the reality of what a "user" actually is. This gap in understanding could result in usability and security concerns.

Related to discretionary access control are POSIX capabilities [9, 12, 13], which can be used to grant additional privileges to specific processes, overriding existing discretionary permissions. This provides a finer-grained alternative to the all-or-nothing superuser privileges required by certain applications. For instance, a web-facing process that requires access to privileged ports has no business overriding file permissions. POSIX capabilities provide an interface for making such distinctions. Despite these benefits, POSIX capabilities have been criticized for adding additional complexity to an increasingly complex Linux permission model [12, 13]. Further, POSIX capabilities do nothing to confine processes—rather, they help to solve the problem of overprivileged processes by limiting the privileges that need to be given to them in the first place.

Namespaces and Cgroups In Linux, *namespaces* and *cgroups* (short for control groups) allow for further confinement of processes by restricting the system resources that a process or group of processes is allowed to access. Namespaces isolate access by providing a process group a private, virtualized naming of a class of resources, such as process IDs, filesystem mountpoints, and user IDs. As of version 5.6, Linux supports eight distinct namespaces, depicted in Table 3.1. Complementary to namespaces, cgroups place limits on *quantities* of system resources that can be used, such as CPU, memory, and block device I/O. Namespaces and cgroups provide fine granularity for limiting the resources that a process or process group can access; however, these are low level mechanisms designed to be used by application developers and higher level frameworks, and thus do not constitute

an adaptive process confinement mechanism by themselves.

Table 3.1: Linux namespaces and what they can be used to isolate.

Namespace	Isolates
PID	Process IDs (PIDs)
Mount	Filesystem mountpoints
Network	Networking stack
UTS	Host and domain names
IPC	Inter-process communication mechanisms
User	User IDs (UIDs) and group IDs (GIDs)
Time	System time
Cgroup	Visibility of cgroup membership

System Call Interposition System call interposition has historically been a very popular process confinement technique, and a number of frameworks exist today for system call interposition on a variety of Unix-like operating systems [5, 30, 31, 47]. System calls define significant portions of the boundary between userspace and kernelspace, and, as such, they capture significant portions of the interface to the operating system's reference monitor [4]; this makes system call interposition a particularly attractive technique for the implementation of fine-grained policy enforcement mechanisms.

One of the earliest forays into system call interposition for process confinement was TRON [7]. Implemented in 1995 for the Unix operating system, TRON provided a kernelspace mechanism for enforcing *protection domains* on userspace processes. A TRON protection domain can be thought of as a set of confined processes, a set of allowed operations, and a *violation handler* which is invoked on policy violations. Processes configure protection domains and then invoke a special `tron_fork` system call to spawn a confined child process. While TRON by itself is not application transparent, it does come with a set of userspace tools to abstract away the configuration of protection domains. Unfortunately, even with these higher level userspace tools, TRON still assumes a certain degree of security expertise in order for a user to properly confine their applications.

Perhaps the most pervasive framework for interposing on system calls is ptrace [31], a process tracing and debugging framework that comes enabled in some form or another on all Unix-like operating systems. While ptrace itself is *not* designed process confinement, some research prototypes [21, 45] have leveraged it in the past. Unfortunately, ptrace is not generally considered production-safe due to its high overhead and buggy interactions with more complex programs such as sendmail. This is especially problematic considering that these are the types of programs that we often wish to confine.

Janus [21, 45] was an early exploration of process confinement using Solaris' version of ptrace. In Solaris, ptrace provides a library call interface into the procfs virtual file system, and allows tracer applications make filtering decisions on behalf of traced processes while interposing on system calls.

Janus was later ported to Linux using a modified version of Linux's `ptrace(2)` system call [45]. In Janus, a supervisor process reads a policy file, and attaches itself to a confined process with ptrace. From there, security-sensitive system calls in the confined application are forwarded to the Janus supervisor process to make a policy decision. This approach, however, adds considerable overhead to confined processes because ptrace requires *multiple* context switches between userspace and kernelspace to coordinate between the tracer and tracee.

To implement its policy language, Janus defines higher level interfaces into various groups of system calls, called *policy modules*. These policy modules can be used to filter groups of related system calls by parameterizing them with the set of allowed actions and system objects. While this abstraction is helpful to group system calls by their related functionality, it does little to help Janus' usability, which is still tightly coupled with the underlying system calls. This makes it difficult for a non-expert user to write effective Janus policy.

[TODO: Talk maybe about Jain and Sekar's work, similar to Janus. If I do talk about this, add it to the summary table]

Anderson published a study in the FreeBSD journal [5] comparing three system call interposition frameworks for three distinct Unix-like operating systems: Linux's seccomp-bpf [18, 27], OpenBSD's pledge [30], and FreeBSD's Capsicum [46, 47]. While these three frameworks all interpose on system calls, they do so with varying degrees of complexity and granularity [5], and so each merits study in its own regard. Figure 3.1 presents an overview of the security and granularity trade-offs in each framework.

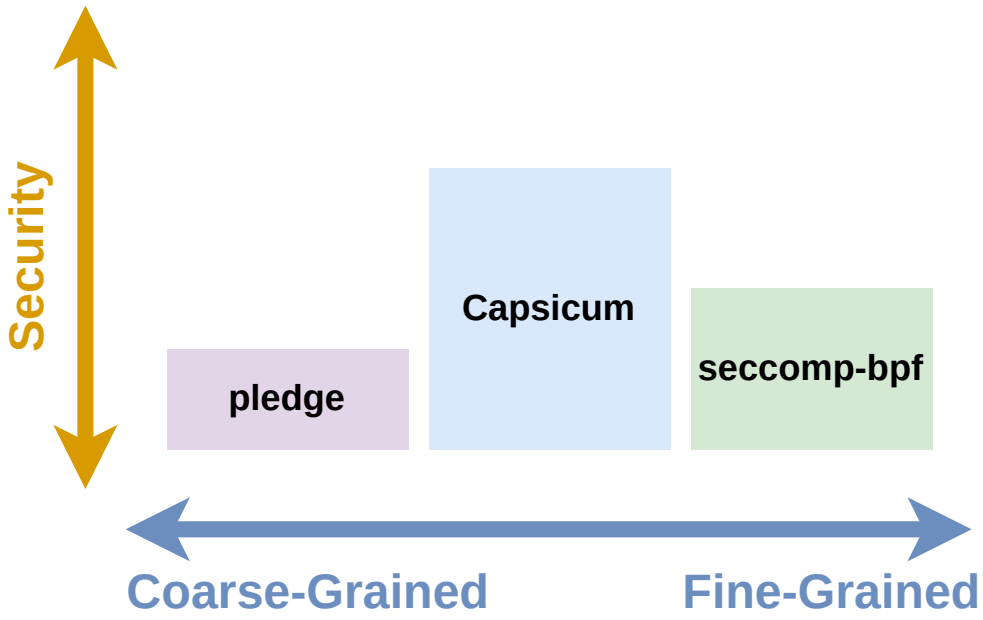

Figure 3.1: Security and granularity trade-offs of pledge, Capsicum, and seccomp-bpf.

In the original Linux seccomp implementation, processes use a special `seccomp(2)` system call to enter a secure computing state. By default, processes that have entered this state are restricted

to performing `read(2)`, `write(2)`, `sigreturn(2)`, and `exit(2)` system calls. Pragmatically, this means that a process could read and write on its open file descriptors, return from invoked signal handlers, and terminate itself. All violations of this policy would result in forced termination. In a 2012 RFC [18], Drewry introduced an extension to seccomp enabling the use of BPF programs[2] for the defining filters on system call arguments. This extension, dubbed seccomp-bpf, enables the creation of fine-grained seccomp policies that filter on system call numbers and arguments, providing a high degree of control to applications that wish to sandbox themselves.

Despite the high degree of control that seccomp-bpf offers to applications, it has severe usability and security concerns, which make it a poor solution for ad-hoc confinement by end users. Classic BPF [29] is a rather arcane bytecode language, and writing classic BPF programs by hand is a task left only to expert users. Further, seccomp-bpf policy is easy to misconfigure, resulting in potential security violations; for instance, a policy that specifies restrictions on the `open(2)` system call but not the `openat(2)` system call can be circumvented entirely. Finally, despite userspace library efforts to abstract away the underlying BPF programs [26], seccomp-bpf remains accessible only to application developers with significant security expertise.

OpenBSD's pledge [30] takes a simpler, coarser-grained approach to system call filtering than seccomp-bpf, instead grouping system calls into high-level semantically meaningful categories, such as `stdio` which includes `read(2)` and `write(2)`, for example [5]. This coarse granularity and simplicity provide increased usability, but come at the expense of expressiveness. For instance, there is no canonical way to distinguish subsets of system call groups or filter system calls by their arguments. Despite its increased usability for developers, pledge still suffers from a lack of application transparency just as seccomp-bpf does, meaning that it is only suitable for use by application developers rather than end users.

Unlike seccomp-bpf and pledge, which apply filtering rules to system calls directly, FreeBSD's Capsicum takes the approach of restricting access to global namespaces via a capability-based implementation [47]. In Capsicum, a process enters *capability mode* using a special `cap_enter` system call. Once in capability mode, access to global namespaces is restricted to the capabilities requested by the process, and these capabilities are inherited across `fork(2)` and `execve(2)` calls. Much like seccomp-bpf and pledge, however, Capsicum is *not* application transparent, and is designed for use by developers rather than end users.

Linux Security Modules The Linux Security Modules (LSM) API [48] provides an extensible security framework for the Linux kernel, allowing for the implementation of powerful kernelspace security mechanisms that can be chained together. LSM works by integrating a series of strategically placed *security hooks* into kernelspace code. These hooks roughly correspond with boundaries for the modification of kernel objects. Multiple security implementations can hook into these LSM hooks and provide callbacks that generate audit logs and make policy decisions. The LSM architecture is

[2]BPF is a special bytecode language, originally implemented for packet filtering in BSD Unix [29]. In seccomp-bpf, it was retrofitted for the purpose of filtering system calls instead.

depicted in Figure 3.2

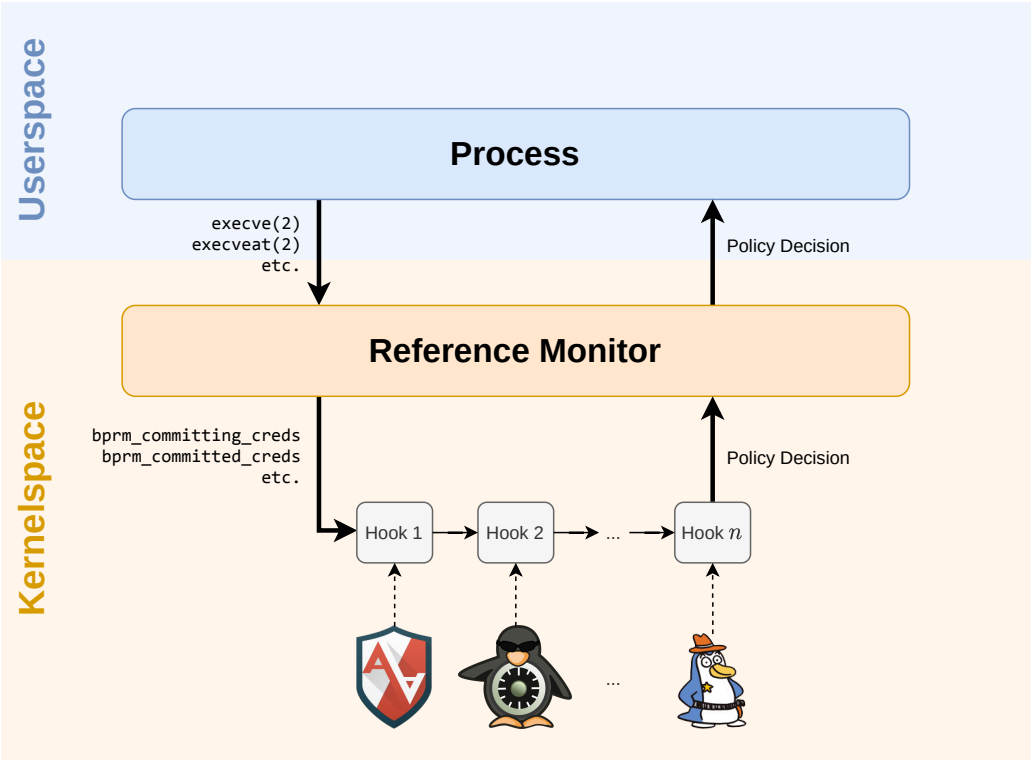

Figure 3.2: The LSM architecture. Note the many-to-many relation between access requests and hook invocations. Multiple LSM hooks may be chained together, incorporating policy from many security mechanisms. All hooks must agree to allow the access or it will be denied.

The LSM API sits at a level of abstraction just above the system call API—a single LSM hook may cover multiple system calls and a single system call may contain multiple such LSM hooks. For instance, the `execve(2)` and `execveat(2)` calls both result in a call to the `bprm_committing_creds` and `bprm_committed_creds` hooks. This provides a nice level of abstraction compared to system-call-based approaches like seccomp-bpf [18, 27] in that a single LSM hook can cover all closely related security events (recall the issue of `open(2)` vs `openat(2)` in seccomp-bpf).

The Linux kernel ships with a number of LSM-based security modules by default. Many such modules implement *mandatory access control* (MAC) schemes, which enable fine-grained access control that can be used to limit the privileges of *all users*—even the superuser. SELinux [37] and AppArmor [14] are two such MAC LSMs, each with its own policy semantics. I discuss each in turn.

SELinux [37] was originally developed by the NSA as a Linux implementation of the Flask [42] security model. Under SELinux, system subjects (users, processes, etc.) and system objects (files, network sockets, etc.) are each assigned corresponding labels. Security policy is then written based on these labels, specifying the allowed access patterns between a particular object type and subject type. SELinux's policy language is famously arcane [35], and despite multiple efforts to introduce automatic policy generation [28, 38], writing and auditing SELinux security policy remains a task for security experts rather than end users. Further, due to the difficulty of writing and auditing the

complex SELinux policy language, there is a natural tendency for human policy authors to err on the side of over-permission, violating the principle of least privilege.

AppArmor (originally called SubDomain) [14] is often touted as a more usable alternative to SELinux, although usability studies have shown that this claim merits scrutiny [35]. Rather than basing security policy on labelling system subjects and objects, AppArmor instead takes the approach of path-based enforcement. Policy is defined in per-application profiles which contain rules specifying what system objects the application is allowed to access. System objects are specified directly rather than being labelled. AppArmor also supports the notion of *changing hats*, wherein a process may change its AppArmor profile under certain conditions specified in the policy. Although AppArmor profiles are more conforming to standard Unix semantics than their SELinux counterparts, users who wish to write AppArmor policy still require a considerable amount of knowledge about operating system security [35].

[TODO: Come back and discuss other LSM if I have time—FBAC-LSM, TOMOYO, SMACK, yama]

3.2 High Level Techniques

In this section, I examine some higher level process confinement techniques that typically employ more than one of the lower level techniques described in the previous section (Section 3.1). Although these solutions generally constitute higher level abstractions, they are not strictly any better than their lower level counterparts, particularly with respect to the adaptive process confinement properties outlined in Section 1.

Containerized Package Management In Linux, a recent trend of *containerized package management* has emerged, allowing users to download applications as packages which are then confined using a combination of lower level techniques exposed by the operating system. Among the most popular of these containerized package management frameworks are Docker [17], Snap [39], and Flatpak [20]. Typically, a package maintainer or application author would write a high-level package manifest declaring the privileges required by the application. This package manifest would then be translated into policy for the underlying process confinement mechanisms (such as those presented in Section 3.1). This architecture is depicted in Figure 3.3.

Despite their widespread adoption, these containerized package management solutions are far from perfect [43]. Due to the coarse granularity of package manifests and high complexity of the underlying policy enforcement mechanisms, policy for containerized applications tends toward gross overpermission. Auditability of policy also suffers due to the disparity between user *expectations* of protection described in package manifests and the actual policy which is enforced in practice. In effect, five or six lines of policy in a package manifest may end up generating thousands of lines of policy across multiple underlying process confinement mechanisms; thus, auditing the generated policy also becomes a seemingly hopeless task, even for seasoned security experts.

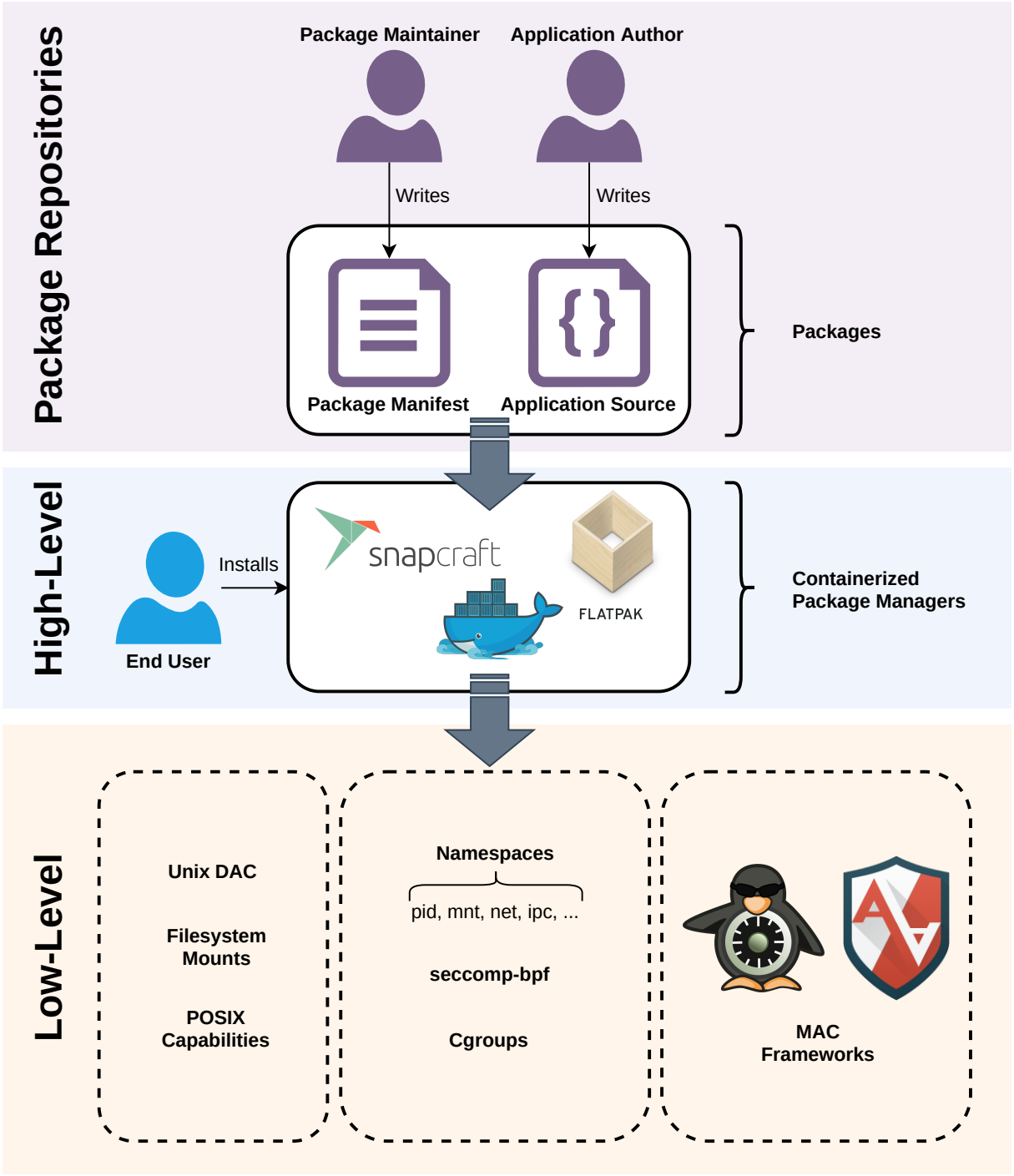

Figure 3.3: The basic architecture of containerized package management solutions for Linux, such as Snapcraft [39], Flatpak [20], and Docker [17]. Package maintainers write high-level, coarse-grained package manifests, which are then compiled into policy for lower-level process confinement mechanisms to enforce.

As a motivating example, consider the package manifest for Snap's Nextcloud[3] package, which includes the Apache `httpd` websever. The package manifest lists the following permissions for Apache `httpd`:

```
confine: strict
/* ... */
apache:
  daemon: simple
  plugs:
  - network
  - network-bind
  - removable-media
```

The result, after policy generation, is 411 seccomp-bpf filters and a 653 line AppArmor policy. The policy is overly permissive, covering a broad scope of capabilities beyond what would be expected based on the three lines of security policy defined in the manifest. For instance, the AppArmor profile allows the execution of over 120 common shell utilities, which is not indicated in the manifest whatsoever. Further, the policy defined in the generated seccomp-bpf policy file often does not align with the policy defined in the generated AppArmor file, adding an extra layer of difficulty to auditing the generated policy.

Ultimately, the problem with containers as an isolation mechanism lies in the complexity of the underlying process confinement mechanisms that they leverage. Namespaces, cgroups, and filesystem mounts are used to virtualize the containers, while complex mechanisms like seccomp, SELinux, and AppArmor are used to enforce least privilege. The simplicity of high-level package manifests belies the complexity underneath, wherein full userlands must be secured for each application. This can often result in a false sense of security, particularly when the underlying process confinement mechanisms are misconfigured, transparently to the end user.

Confinement with Wrapper Sandboxes Separate but related to containerized application management are *wrapper sandboxes*. Like containers, these wrappers use a combination of multiple lower level process confinement techniques to achieve an end result. The user typically executes a wrapper application which sets up the sandbox before launching the target application (e.g. `generic-wrapper-application firefox`). Some examples of these wrappers include Firejail [19], Bubblewrap [8], and MBox [24].

Even very early on in the development of Unix process confinement mechanisms, it was recognized that wrapper applications can provide significant usability benefits by abstracting away the underlying details of process confinement. For instance, TRON [7] provided a wrapper application for its protection-domain-based enforcement back in 1995. While these wrapper applications often do come with usability and transparency benefits, they suffer from auditability and robustness issues not dissimilar from the containerized package managers discussed above.

Firejail [19] uses seccomp-bpf and Linux namespaces to confine applications. Recent versions

[3]Available: https://snapcraft.io/nextcloud

also include a generic AppArmor profile to provide an extra layer of basic protection. As with other high level process confinement mechanisms, policy is defined using per-application profiles which are then compiled down to the underlying policy mechanisms. While these policy files are quite high level and designed to be used out the box (on Arch Linux, the latest version of Firejail comes with over 1000 pre-configured profiles), their auditability and robustness suffer due to the coarse-granularity of rules. For instance, Firejail takes an all-or-nothing approach to file access, specified using either `whitelist`, `blacklist`, or `noblacklist`. It is possible to combine all three keywords on a shared pathname, leading to results that may be unpredictable for end users[4].

Kim and Zeldovich take a rather different approach with Mbox [24]; instead of using policy profiles, access in Mbox is specified using coarse-grained flags passed to a wrapper application. To confine applications, Mbox leverages a seccomp-bpf to filter and interpose on system calls, and ptrace to *re-write* system call arguments. Applications are confined to a sandboxed filesystem by re-writing the arguments to the `open(2)`, `read(2)`, and `write(2)` system calls. Since Mbox policy is expressed using command line arguments, auditing Mbox is impossible without actually reading its source code. Further, a user requires sufficient knowledge to know which command line arguments should be used with which application.

Like Mbox, Bubblewrap [8] confines applications using command line arguments. Bubblewrap confines applications using a combination of filesystem mounts, Linux namespaces and seccomp-bpf rules, and is designed to work cooperatively with other container-based approaches like Flatpak [20]. Just as with Mbox, the auditability and usability of Bubblewrap suffers due to the advanced knowledge required to run an application using the appropriate confinement arguments, which may end up being overly permissive regardless.

3.3 (Semi-)Automated Policy Generation

Usability and transparency issues represent a common theme among several of the process confinement mechanisms I have discussed thus far. This is particularly true for highly complex, low-level mechanisms like SELinux [37], AppArmor [14], and seccomp-bpf [18, 27]. It is tempting to turn toward the notion of *automatic policy generation* to alleviate these issues.

Both SELinux and AppArmor include mechanisms for automatic policy generation. AppArmor's `aa-easyprof` [1] guides the user through policy generation, and supplementary templates and abstractions can be provided to bootstrap the policy generation process. However, policy generated using this method tends toward overpermission and its quality is highly dependent on the quality of the provided policy templates [1]. AppArmor also supports two lower-level policy generation tools. `aa-genprof` [2] generates policy using AppArmor's audit logs, while `aa-logprof` [3] does the same thing with guided input from the user. Both these approaches have the advantage of higher transparency to the end user, but may generate more restrictive policy which could require manual

[4]See this GitHub issue for an example: https://github.com/netblue30/firejail/issues/1569

modification to function.

SELinux policy generation was originally supported via the `audit2allow` [38] command line tool. This tool functions similarly to `aa-genprof` [2] above, automatically generating policy with the help of SELinux's audit logs. Sniffen *et al.*recognized the need for improving upon the existing policy generation in SELinux and introduced a guided policy generation approach, `polgen` [40]. `polgen` semi-automates the policy generation process using modified version of the `strace`[5] command line utility which interposes on system calls and analyzes patterns in their arguments to infer policy. Policy is expressed in an intermediate representation language called PSL, which is then fed into a modelling program which constructs an information flow graph and a type generation program to generate new labels for system objects. In the end, the resulting PSL policy is compiled into actual SELinux policy to be installed on the system. Unfortunately, this approach does little to alleviate SELinux's primary usability concerns, as the generated policy is still tied down to SELinux's arcane labelling and type enforcement mechanisms.

The 2006 introduction of the SELinux reference policy [32] refactored existing SELinux sample policy into re-usable modules that could be plugged into new policy files using higher level interfaces. This motivated MacMillan to develop Madison [28], yet another take on SELinux policy generation. Rather than re-inventing the wheel, Madison was designed to be complementary to existing policy generation mechanisms. The idea was that Madison would be used to generate the *reference policy* itself, which could then be integrated with new policy files using existing tools like `polgen` or by hand. Miranda's approach does help to abstract away the underlying SELinux implementation details, but ultimately falls short of its goal of improving usability. Users still need to understand the underlying SELinux policy language if they are to have any hope of auditing or modifying the generated reference policy files.

[TODO: maybe TOMOYO actually belongs here?]

While SELinux and AppArmor included policy generation as an afterthought, other approaches to policy generation have taken an alternative approach, building the initial process confinement mechanism with policy generation in mind. Systrace [33]

[TODO: Inoue Java policy]

While policy generation can indeed help with transparency issues in policy authorship, it does not necessarily fully alleviate usability concerns. So long as the underlying confinement language remains complex, it will be difficult for a user to audit the generated policy. Manual modification of the generated policy also remains a challenge here; it would be desirable to make this easy so as to enable ad hoc process confinement by end users and painless debugging of overly restrictive auto-generated policies.

[5] `strace` uses the ptrace system call under the hood.

3.4 (Semi-)Automated Policy Audit

4 Adaptive Analysis

[Intro]

A *maladaptive* process confinement mechanism is one which does not cleanly fit the majority of the adaptive properties outlined in Section 1 (items P1–P5). For example, a process confinement mechanism with high reconfigurability and low adoption effort, but with low robustness to attacker innovation, low transparency, and low usability would constitute a maladaptive approach.

[Semi-Adaptive Description]

Table 4.1: The adaptive evaluation for process confinement techniques discussed in this section. Since Unix DAC is not a process confinement mechanism in its own right, it is omitted here. Note that scores for lower level mechanisms *do not* consider the merits of the higher level mechanisms that make use of them.

Mechanism	P1 Robustness	P2 Adoptability	P3 Reconfigurability	P4 Transparency	P5 Usability	Score (max 5)
POSIX Capabilities	○	●	○	◐	○	TODO
Namespaces + Cgroups	◐	●	○	○	○	TODO
ptrace	○	○	○	○	○	TODO
Janus	◐	○	○	◐	○	TODO
seccomp-bpf	◐	●	○	○	○	TODO
capsicum	◐	●	○	○	○	TODO
pledge	○	●	○	○	○	TODO
SELinux	○	○	○	○	○	TODO
AppArmor	○	○	○	○	○	TODO
Docker	○	●	○	◐	◐	TODO
Snap	○	●	○	◐	◐	TODO
Flatpak	○	●	○	◐	◐	TODO
Firejail	○	◐	○	◐	◐	TODO
Bubblewrap	○	◐	○	○	○	TODO
Mbox	○	○	○	○	○	TODO

5 Towards Truly Adaptive Process Confinement

5.1 Anomaly Detection Techniques

5.2 Extended BPF

[bpfbox]

6 Conclusion

References

[1] *aa-easyprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-easyprof.8.html.

[2] *aa-genprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-genprof.8.html.

[3] *aa-logprof(8)*, Linux user's manual. [Online]. Available: https://manpages.ubuntu.com/manpages/precise/man8/aa-logprof.8.html.

[4] J. Anderson, "Computer Security Technology Planning Study," Tech. Rep. ESD-TR-73-51, 1973, Section 4.1.1. [Online]. Available: https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/ande72.pdf.

[5] J. Anderson, "A Comparison of Unix Sandboxing Techniques," *FreeBSD Journal*, 2017. [Online]. Available: http://www.engr.mun.ca/~anderson/publications/2017/sandbox-comparison.pdf.

[6] L. E. Beegle, "Rootkits and Their Effects on Information Security," *Information Systems Security*, vol. 16, no. 3, pp. 164–176, 2007. DOI: 10.1080/10658980701402049.

[7] A. Berman, V. Bourassa, and E. Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System," in *Proceedings of the USENIX 1995 Technical Conference*, The USENIX Association, 1995, pp. 165–175. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9149&rep=rep1&type=pdf.

[8] Bubblewrap, *Bubblewrap*, 2020. [Online]. Available: https://github.com/containers/bubblewrap (visited on 10/25/2020).

[9] *capabilities(7)*, Linux user's manual. [Online]. Available: https://linux.die.net/man/7/capabilities.

[10] F. B. Cohen, "A Secure World-Wide-Web Daemon," *Comput. Secur.*, vol. 15, no. 8, pp. 707–724, 1996. DOI: 10.1016/S0167-4048(96)00009-0.

[11] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-Sharing System," in *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, ser. AIEE-IRE '62 (Spring), San Francisco, California: Association for Computing Machinery, 1962, pp. 335–344, ISBN: 9781450378758. DOI: 10.1145/1460833.1460871.

[12] J. Corbet, "A bid to resurrect Linux capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/199004/.

[13] J. Corbet, "File-based capabilities," *LWN*, 2006. [Online]. Available: https://lwn.net/Articles/211883/.

[14] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14st Large Installation Systems Administration Conference (LISA)*, New Orleans, LA, United States: USENIX Association, 2000. [Online]. Available: https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf.

[15] A. Crowell, B. H. Ng, E. Fernandes, and A. Prakash, "The Confinement Problem: 40 Years Later," *Journal of Information Processing Systems*, vol. 9, no. 2, pp. 189–204, 2013. DOI: 10.3745/JIPS.2013.9.2.189. [Online]. Available: http://jips-k.org/journals/jips/digital-library/manuscript/file/22579/JIPS-2013-9-2-189.pdf.

[16] Discord, *Discord Privacy Policy*. [Online]. Available: https://discord.com/privacy (visited on 10/25/2020).

[17] Docker, *Docker Security*, 2020. [Online]. Available: https://docs.docker.com/engine/security/security (visited on 10/25/2020).

[18] W. Drewry, "Dynamic seccomp policies (using BPF filters)," Internet RFC, 2012. [Online]. Available: https://lwn.net/Articles/475019/.

[19] Firejail, *Firejail*, 2020. [Online]. Available: https://firejail.wordpress.com (visited on 10/25/2020).

[20] Flatpak, *Sandbox Permissions*, 2020. [Online]. Available: https://docs.flatpak.org/en/latest/sandbox-permissions.html (visited on 10/25/2020).

[21] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, "A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker)," in *Proceedings of the Sixth USENIX UNIX Security Symposium*, The USENIX Association, 1996. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf.

[22] Google, *Android Security Features*, Android security documentation. [Online]. Available: https://source.android.com/security/features (visited on 10/26/2020).

[23] R. M. Graham, "Protection in an information processing utility," *Communications of the ACM*, vol. 11, no. 5, pp. 365–369, 1968, ISSN: 0001-0782. DOI: 10.1145/363095.363146.

[24] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, USENIX Association, 2013, pp. 139–144. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim.

[25] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973, ISSN: 0001-0782. DOI: 10.1145/362375.362389.

[26] libseccomp authors, *libseccomp*. [Online]. Available: https://github.com/seccomp/libseccomp (visited on 10/27/2020).

[27] Linux, *Seccomp BPF (SECure COMPuting with filters)*, Linux kernel documentation. [Online]. Available: https://static.lwn.net/kerneldoc/userspace-api/seccomp_filter.html (visited on 10/27/2020).

[28] K. MacMillan, "Madison: A new approach to policy generation," in *SELinux Symposium*, 2007. [Online]. Available: http://selinuxsymposium.org/2007/papers/08-polgen.pdf.

[29] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *USENIX Winter*, vol. 93, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[30] OpenBSD, *pledge(2)*, OpenBSD user's manual. [Online]. Available: https://man.openbsd.org/pledge.2.

[31] P. Padala, "Playing with ptrace, Part I," *Linux Journal*, vol. 2002, no. 103, p. 5, 2002. [Online]. Available: https://www.linuxjournal.com/article/6100.

[32] C. J. PeBenito, F. Mayer, and K. MacMillan, "Reference Policy for Security Enhanced Linux," in *SELinux Symposium*, 2006. [Online]. Available: http://selinuxsymposium.org/2006/papers/05-refpol.pdf.

[33] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 13th USENIX UNIX Security Symposium*, The USENIX Association, 2003. [Online]. Available: http://citi.umich.edu/u/provos/papers/systrace.pdf.

[34] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," in *Proceedings of the Fourth ACM Symposium on Operating System Principles*, ser. SOSP '73, New York, NY, USA: Association for Computing Machinery, 1973, p. 27, ISBN: 9781450373746. DOI: 10.1145/800009.808045.

[35] Z. C. Schreuders, T. J. McGill, and C. Payne, "Towards Usable Application-Oriented Access Controls," in *International Journal of Information Security and Privacy*, vol. 6, 2012, pp. 57–76. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.963.860&rep=rep1&type=pdf.

[36] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561, ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/1315245.1315313.

[37] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux security module," 43, vol. 1, 2001, p. 139. [Online]. Available: https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf.

[38] J. R. Smith, Y. Nakamura, and D. Walsh, *audit2allow(1)*, Linux user's manual. [Online]. Available: http://linux.die.net/man/1/audit2allow.

[39] Snapcraft, *Security policy and sandboxing*, 2020. [Online]. Available: https://snapcraft.io/docs/security-sandboxing (visited on 10/25/2020).

[40] B. T. Sniffen, D. R. Harris, and J. D. Ramsdell, "Guided policy generation for application authors," in *SELinux Symposium*, 2006. [Online]. Available: http://gelit.ch/td/SELinux/Publications/Mitre_Tools.pdf.

[41] E. H. Spafford, "The Internet Worm Incident," in *ESEC '89, 2nd European Software Engineering Conference, University of Warwick, Coventry, UK, September 11-15, 1989, Proceedings*, ser. Lecture Notes in Computer Science, vol. 387, Springer, 1989, pp. 446–468. DOI: 10.1007/3-540-51635-2_54.

[42] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. G. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*, USENIX Association, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[43] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. DOI: 10.1109/ACCESS.2019.2911732.

[44] US Department of Defense, "Trusted Computer System Evaluation Criteria," DOD Standard DOD 5200.58-STD, 1983.

[45] D. A. Wagner, "Janus: An Approach for Confinement of Untrusted Applications," M.S. thesis, University of California, Berkeley, 1999. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/CSD-99-1056.pdf.

[46] R. N. M. Watson and J. Anderson, *capsicum(4)*, FreeBSD user's manual. [Online]. Available: https://www.unix.com/man-page/freebsd/4/capsicum/.

[47] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010*, USENIX Association, 2010, pp. 29–46. [Online]. Available: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf.

[48] 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, San Francisco, CA, USA, August 5-9, 2002*, USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html.