Towards Adaptive Process Confinement Mechanisms

COMP5900I Literature Review

William Findlay

October 26, 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 [4, 12, 14]. 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 [14], 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 [13]. 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 [13], it remains largely unsolved to date [7]. 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. High 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. Low adoption effort. 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. High 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. High 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. High 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 discusses maladaptive process confinement solutions, while Section 4 presents semi-adaptive approaches that fail to meet the mark of a truly adaptive process confinement mechanism. 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 [3]. 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 [15], 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 [17]. 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 [8] 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* [1] 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 [13]) 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, 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 [8] 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 [7], and so the exploration of new solutions is necessary.

3 Maladaptive Process Confinement Approaches

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

[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.

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.

Unfortunately, maladaptive techniques cover the majority or the process confinement literature, but many are widely used and studying them can provide good insight into what an adaptive or semi-adaptive approach might do differently in practice. To further compartmentalize this study, I categorize maladaptive approaches into the lower-level techniques (e.g. at the operating system kernel or application source code levels) and the higher-level techniques that often build upon them.

Table 3.1: Test

Mechanism	Classification	P1 Robustness	P1 Robustness	P1 Robustness	P1 Robustness	P1 Robustness
Foo	Bar	●	●	●	●	●

3.1 Low Level Techniques

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 [18], a discretionary access control system mechanism system objects (e.g. files) by their respective owners, and allows resource owners to grant access to other users at their discretion. Typically, these systems 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 and GID to *each* application on the system [11].

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 nothing preventing a 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 [2, 5, 6], 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 that would be 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. Despites these benefits, POSIX capabilities have been criticized for adding additional complexity to an increasingly complex Linux permission model [5, 6].

[Namespaces, Cgroups]
[seccomp-bpf, capsicum, pledge, ptrace]
[Linux MAC: LSM Hooks, SELinux, AppArmor, TOMOYO, yama]

3.2 High Level Techniques

In Linux, a recent trend of *containerized package management* has emerged. [Snapcraft, Docker, Flatpak]

[Firejail, Bubblewrap, mbox]
[Janus]

4 Semi-Adaptive Process Confinement Approaches

4.1 Automated Policy Generation

4.2 Automated Policy Auditing

5 Towards Truly Adaptive Process Confinement

5.1 Anomaly Detection Techniques

5.2 Extended BPF

[bpfbox]

6 Conclusion

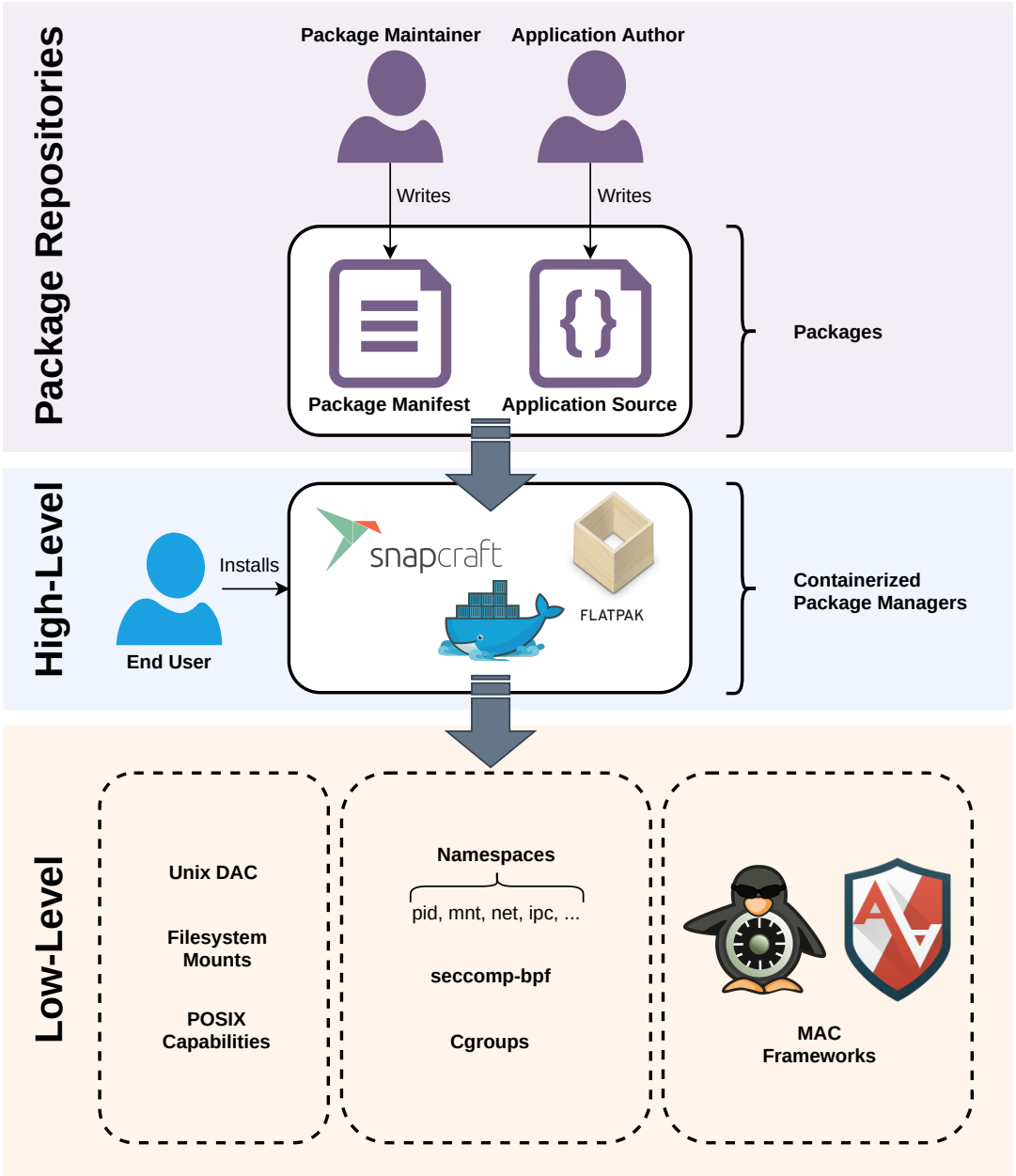

Figure 3.1: The basic architecture of containerized package management solutions for Linux, such as Snapcraft [16], Flatpak [10], and Docker [9]. Package maintainers write high-level, coarse-grained package manifests, which are then compiled into policy for lower-level process confinement mechanisms to enforce.

References

[1] 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.

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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