Leash: A Transparent Capability-Based Sandboxing Supervisor for Unix

Mahya Soleimani Jadidi[a] and Jonathan Anderson[b]
Department of Electrical and Computer Engineering, Memorial University of Newfoundland, NL, Canada

Keywords: Application Sandboxing, System Call Filtering, Capability-Based Sandboxing, Application Compartmentalization.

Abstract: In computer security, the principle of least privileges or denial by default is a practical approach to mitigate the risk against potential attacks. However, providing least-privileged applications is a challenge without source code modification, system privilege, or configuration changes. In this paper, we introduce Leash, a transparent application sandboxing supervisor for Unix systems designed based on FreeBSD's Capsicum framework. Leash provides required resources to programs based on sandbox restrictions and policies predefined by the user without requiring root privilege. The approach is transparent to the code and the user, eliminating the need for any source code modification and deep knowledge about the underlying security framework. We evaluated this system by sandboxing a set of widely used Unix utilities and real-world installer scripts. Leash is designed to be expandable for becoming a general-purpose sandboxing service for Unix. Our evaluations show that the system achieves robust security while maintaining efficient performance.

1 INTRODUCTION

The principle of least privilege, also known as "deny by default", is a fundamental strategy for mitigating security risks. For applications exposed to untrusted networks, enforcing least-privilege execution enhances both the security of the application and the underlying host system. However, implementing this principle in applications is challenging, especially without extensive code modifications or system configuration changes.

In Unix, installing many applications involves downloading files and executing shell scripts, often with elevated privileges (i.e., root). This includes development tools, libraries, and system management utilities. Installer scripts are attractive targets for malicious actors, particularly in open-source software. Thus, protecting users and developers from harmful installation scripts and supply chain attacks is essential.

In this paper, we present Leash, a system designed for transparent sandboxing of applications in Unix environments without requiring root privilege. Leash offers a lightweight sandboxing mechanism, that runs an application with least privileges. It is built on Capsicum, a sandboxing framework for FreeBSD. Leash establishes a user-level sandboxing environment with Capsicum *capabilities*, granting only the essential access privileges to the executing process. Leash provides two levels of transparency:

- Code transparency: Leash does not require modifying the target's source code.
- User transparency: Leash does not require users to have in-depth knowledge of Capsicum.

This paper is organized as follows: Section 2 provides essential background information. Section 3 discusses key challenges in designing a transparent sandboxing system. Section 4 describes the internal design, including the framework and policies. Section 5 outlines our evaluation methodology and discusses key findings. Related work is reviewed in section 6. Future work is covered in section 7, and section 8 summarizes this study, including the conclusions.

2 BACKGROUND

Leash is designed based on FreeBSD, which features a *capability-based* framework in its kernel called Capsicum (Watson et al., 2010). Capabilities are unforgeable tokens of authority that authorize process access to resources (Fabry, 1974; Dennis and van Horn, 1975; Anderson, 1972).

[a] https://orcid.org/0000-0002-7897-410X
[b] https://orcid.org/0000-0002-7352-6463

Jadidi, M. S. and Anderson, J.
Leash: A Transparent Capability-Based Sandboxing Supervisor for Unix.
DOI: 10.5220/0013186700003899
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025) - Volume 2*, pages 542-551
ISBN: 978-989-758-735-1; ISSN: 2184-4356
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

In capability-based systems, capabilities are tied to resources, such as files. They are also known as *object capabilities*. Different processes can have different access rights to the same resource. While processes can pass capabilities to one another, they cannot elevate them. Capabilities are unforgeable, allowing the platform or kernel to verify their validity and prevent unauthorized changes. So, capability-based systems can enforce the principle of least privilege, enabling fine-grained, flexible, and dynamic permission management.

2.1 Capsicum

In Capsicum, capabilities are implemented in the structure of file descriptors that reference system resources. In Capsicum's capability mode, entered by calling `cap_enter()`, access to global namespaces will fail. Processes can only access resources for which they have capabilities. This restriction will be applied to the running process and its further descendants.

Capsicum has three categories of system calls: disallowed functions, conditionally allowed functions, and allowed functions. System calls that try to gain new access to global namespaces of the system, such as `open(2)`, `connect(2)`, `mkdir(2)`, and `wait4(3)`, are all disallowed. Conditionally allowed functions are safe functions that can gain unsafe privileges under some circumstances. For example, calling `open(2)` with `AT_FDCWD` set as the file descriptor tries to open a file in the current working directory.

Capsicum offers a secure, restricted environment to enhance application safety. However, adapting to Capsicum's sandbox requires significant changes to source code. Developers must replace unsafe global namespaces access with least-privileged ones, which also requires in-depth knowledge of the operating system and Capsicum. Section 4 outlines our approach in addressing these challenges, and describes how Leash balanced security and functionality to enable more applications to run safely within Capsicum's sandbox.

2.2 Linux Capabilities

Linux uses the term "capability" to refer to distinct units of privilege, which subdivide the traditional superuser (root) privileges into a more granular model. This allows root users to be granted different levels of access to critical system resources (Linux man-pages project, 2024; Linux man-pages project, 2022). For example, capabilities like `CAP_AUDIT_READ`, `CAP_KILL`, and `CAP_CHOWN` enable a process to read audit logs, terminate other processes, and change file ownership, respectively. Setting Linux capabilities needs root privilege. This is an entirely distinct concept from the object capabilities described above, which refer to individual objects, such as files, and do not require root privilege to use.

3 CHALLENGES IN DESIGNING SANDBOXING SYSTEMS

Designing a sandboxing system is challenging. As the primary concern, the sandboxing system should isolate the target program by making an adequately confined environment that is not circumventable. As described in section 6.1, existing sandboxing tools are either not capable of providing such security or need invasive modifications and complicated configurations. Many existing tools and OS features provide isolation by restricting the program to a specific file system hierarchy. However, file system restriction is not enough for sandboxing applications that can access other resources, such as the network.

Many existing applications have not been designed with security in mind. Securing such applications with existing sandboxing features is either impossible or needs modifications on the source code, which is a challenge for those with massive code bases. Hence, transparent sandboxing is required.

Here are the most important problems we resolved during the design and implementation of Leash:

1. Process management: The target process should be able to fork, execute, and wait for its descendants. The significant point is that the forked or executed processes should also operate in the sandbox mode. The challenge is that the supervisor cannot control the target's descendants because they are not processes forked by the supervisor directly. However, the supervisor can be informed of such function calls to prepare for future requests and an under-control environment. Section 4.1 explains more details about this issue and our solution to control it.

2. Remote connections: One challenge in controlling the target's connections is the complexity of connecting to remote servers, such as connecting to the internet. DNS resolving and internet communications can take more time, and managing TCP/IP communication should be handled on the server side. The inter-process communication (IPC) between the supervisor and the target process should be carefully designed to take these considerations into account.

3. Circular dependencies in GNU Standard C Library (libc): in section 4.4, we described how we handled transparent behaviour changing through function interposition. However, the design of an interposition technique can be limited based on various reasons. One of these challenges is the existence of circular dependencies in the targeted set of functions.
4. Process context limitations: Each process has a distinct context and a set of environment variables, which may be inherited from its parent process. One management problem for the supervisor process is the changes that happen to this context, such as changing the current working directory or re-initialization of global variables that happens after calling `exec()` functions. The supervisor process needs to be updated about these changes to update the assumptions for further required decision-making. We handled these cases in the interposition library described in section 4.4.

4 DESIGN AND IMPLEMENTATION

In this section, we explain the internal design of Leash, and development challenges. Figure 1 shows an abstraction of Leash's main modules integrated to form a system for executing the client program in Capsicum's sandbox mode. Our design is divided into four main modules: the supervisor server, the policy management module, the command handler, and the interposition library.

The supervisor process calls the policy manager to initialize the policies defined for the client program specifically. The loaded definitions will be used to provide necessary environment variables and resources, including pre-loaded files and libraries, prior to executing the target(client) program. The supervisor establishes an IPC[1] mechanism between itself and the client for further expected communications. Finally, the supervisor executes the client program in the sandbox mode. From this point on, every disallowed function will be either interposed and proxied to the supervisor for decision-making or fail due to capability violations.

4.1 Supervisor Server

The main thread of the supervisor process is responsible for providing required environment variables, pre-opening the necessary libraries and a pre-established connection for communication between the supervisor and the client processes, loading the policies, and serving commands proxied to the command handler threads. Figure 2 demonstrates this procedure.

[1]Inter Process Communication

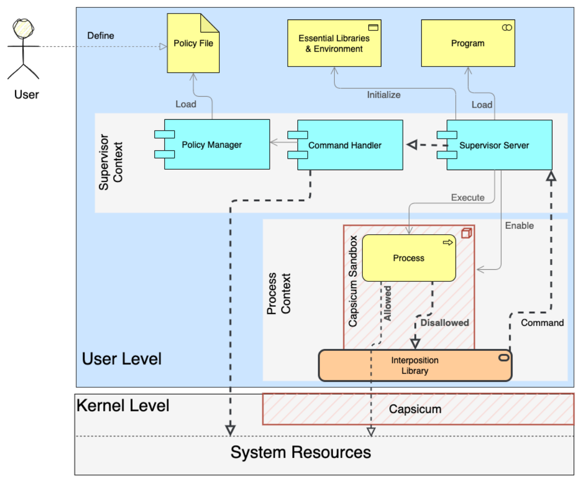

Figure 1: Leash's different modules and their integration.

After the supervisor fetches all the required information for executing the target program, it forks. The parent process continues to initiate a server thread, and the child process goes on and enters the sandbox mode by calling `cap_enter()` and executes the target program. From this point on, every disallowed function will be either interposed or failed. The disallowed system calls that are not supported in the current design will fail due to capability violations. Those supported will be interposed by the interposition library, which determines whether to wrap or proxy them. Table 1 lists all the supported system calls in the current design. The proxied requests will be considered as *commands*, described in section 4.2.

The server can handle parallel commands as demonstrated in fig. 2. The reason is that the client process can fork in capability mode, and the descendants are as restricted as their parent processes. The supervisor server initializes a new connection and command handler for each new process. Hence, Leash can support the target and its descendants separately, also handles process management requests.

4.2 Command Handler

When the interposition library intercepts a disallowed function call, it will be proxied to the Supervisor Server if it cannot be handled on the client's side. We refer to a proxied call as a command delegated to the server. The Command Handler module extracts a new command from the received information. To manage the request, the newly made command checks the request with the Policy Manager module, interacts with

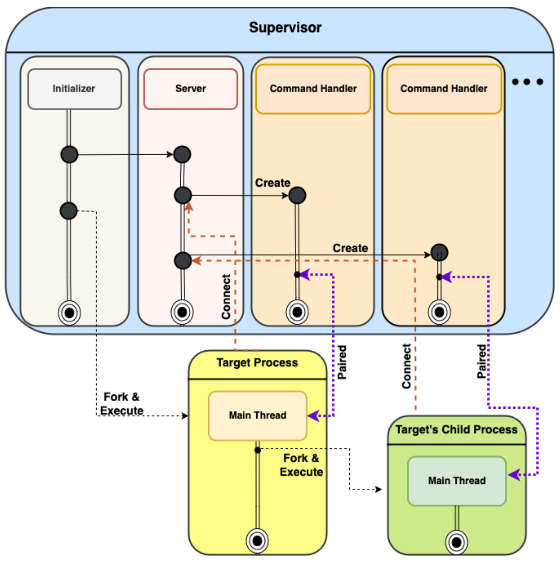

Figure 2: The supervisor multi-threaded request handling.

Listing 1: Sample policy specification for sandboxing curl(8) by Leash.

```
paths {
   "./curl_cmd.sh":
        open|read|write|exec|lookup|seek;
   "/dev/crypto":
        open|create|read|write|exec|lookup|seek;
   "/usr/bin/curl": open|read;
   others: ask;
}

net {
  "172.217.13.100": connect;
  others: ask;
}

sysctl {
   "net.inet.ip.forwarding": read;
   "hw.cachesize": read;
   others: deny;
}
```

the user if needed, and will do the changes required on behalf of the client process if:

1. The request does not conflict with any of the predefined policies.
2. The user allows the program to proceed if asked.

When the command is resolved and done, the output will be returned to the client as feedback, and the serving thread will be prepared to handle the following probable command. The supervisor process does not change the functionality of received commands but may update policies based on served requests.

4.3 Policy Management

Leash is designed to be an interactive system depending on how it is specified by the user as policies. The policies are meant to be defined in a file written in UCL format (UCL Documentation, 2014). Leash loads the policy file, defines the required environment variables, and pre-opens necessary libraries before executing the target program in the sandbox.

The policy file is expected to include three main sections for accessing pathnames, networking, and system control requests. These sections should be labelled as *path*, *net*, and *sysctl*, respectively. Each section of the policy contains the resources that are addressed by a unique name, such as an absolute address to a file. The minimum required access privileges are specified following the resource name. When a request is received from the client process, the behaviour will be determined based on the loaded policies. Each section should include an entry labelled as "others", which determines the behaviour of the system in case the resource is not listed in the policy file. Hence, we can define different default actions for categories of resources. When the target process tries to access to unspecified resources, the system behaves based on the policy defined for "others" in the corresponding section, as explained below:

- ALLOW: The supervisor allows the command to proceed with the command.
- DENY: The supervisor denies the command and returns a failure message to the target process.
- ABORT: The supervisor kills the client process.
- ASK: The supervisor asks the user if the request is allowed to proceed.

Listing 1 includes a sample used for curl(8) command to download a file. At the current state, we need to address every single required pathname, explicit IP addresses, and system control variable names. However, as described in section 7, one of the future work to do is making policy definition more convenient in Leash.

4.4 Function Interposition

The interposition library is one of the resources provided by the supervisor process for the target program. The supervisor sets the `LD_PRELOAD` environment variable to address the interposition library. By setting the `LD_PRELOAD` environment variable, the priority of the runtime linker (RTLD) for resolving the functions changes. So, in the procedure of function resolving, the pre-loaded library will be considered first. This mechanism provides an opportunity for intercepting disallowed functions in Capsicum's capability mode at runtime. In the interposition library, disallowed functions will be replaced by either a safer version or a proxy function. The proxy functions send the call information as a command to the supervisor and extract the result from its response. The interposition library returns all the expected results and data required to maintain the functionality of the function calls and does not change the functionality or returned data.

5 EVALUATION

In this section, we describe our methodology and observed results of evaluating the behaviour and performance of Leash on real-world examples. We selected the installer script of Ruby Version Manager(RVM) as our primary target (RVM, 2009). To install RVM, the user needs to download the installer script and execute it using `Bash(1)` command, as recommended by the providers. To succeed, every command the script calls should inevitably be sandboxed. Therefore, along with RVM's installer script, we sandboxed other utilities including Curl, Fetch, Gtar, and Gzip. Moreover, 21 of other Unix' simple commands were sandboxed as shown in Table 3.

5.1 Platform

Leash is designed and implemented on FreeBSD 13.2-RELEASE as the base OS. However, to achieve our objectives, we applied some modifications to the OS source code, including the kernel. We added the system call `wait6(2)` to the allowed functions in capability mode for the cases where it is called on any child processes. The evaluations were done on an amd64-based machine with an 8-core CPU of the model Intel(R) Xeon(R) CPU E3-1240 v5 @ 3.50GHz. To obtain the actual time taken for the executed commands and the memory usage, we used FreeBSD's `time(1)` and `ps(1)` commands.

5.2 Discussion

As mentioned, to install RVM in the sandbox, we sandboxed all the utilities this installer called. To achieve this goal, we interposed all the disallowed functions called by the installer scripts or the corresponding commands, which resulted in the system calls categorized in Table 1. Along with the main system calls, their other versions that were limited in capability mode needed to intercept. These versions include internal definitions of the disallowed functions such as `_open()`, and the versions that are usually

Table 1: Interposed system calls for sandboxing RVM's installer script and the commands it called.

Purpose	System calls	
	Main Version	Other Versions
File System Access	open()	openat(), _open()
	stat()	fstatat(), lstat()
	statfs()	
	readlink()	
	mkdir()	mkdirat()
	chdir()	fchdir()
	eaccess()	eaccessat()
	unlink()	unlinkat()
	rename()	_rename()
	utimes()	utimensat(), _utimes()
	chmod()	fchmodat(), _chmod()
	symlink()	symlinkat()
System Control	sysctl()	__sysctl()
Process Management	execve()	_execve()
	fork()	vfork(), _fork()
	wait4()	waitpid(), _wait4()
Networking	connect()	_connect()

allowed with some exceptions such as `openat(2)` called with `AT_CWDFD` as the first argument.

The performance overhead added by executing programs under Leash is directly related to the number of intercepted system calls, especially those proxied to the server. So, the interposition library will add most of the overhead. This overhead can increase depending on the functionality of the target program. For example, remote connections or massive manipulations on the file system can result in more than one intercepted system call and higher overhead. However, our observations show that this overhead is practically acceptable and tolerable.

Tables 2 and 3 lists applications and Unix commands executed in Leash's sandbox. Table 2 separates the heavier applications in terms of the volume of calls, interaction with the system, and longer execution time and provides more details about the observed performance overhead. Table 3 demonstrates the commands that are lighter, and their execution time and overhead were at the scale of milliseconds. The chart presented in fig. 3 compares the performance of the native command vs. the sandboxed execution. Although the user cannot sense the sandboxing overhead on these utilities, our measurements show that the slowdown that happened to them by sandboxing is more significant than the applications listed in Table 2. This observation shows that the cost of running the supervisor program, runtime interposition, proxying commands, and supervisor command handling is more observable with fewer system calls. While in programs with larger code bases and more functionalities, the cost of allowed function pervades the sandboxing overhead.

The most time-consuming mode of Leash is the one in which every single disallowed function will

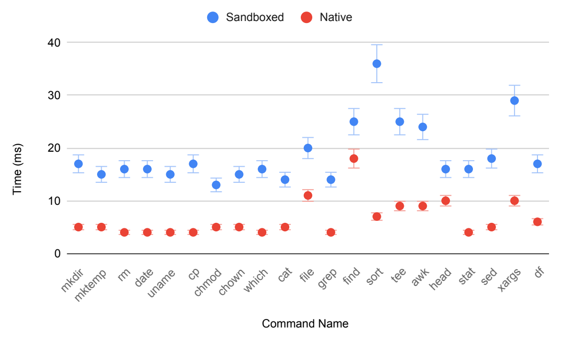

Figure 3: Unix utilities executed in Leash's sandbox and their performance overhead.

Table 2: The performance of the sandboxed applications and utilities under Leash's supervision.

Application	Interposed	Average Time (Second)		Slowdown
		Original	Sandboxed	
RVM	17433	5.68	6.25	10%
curl	31	0.30	0.35	16%
fetch	22	0.24	0.25	4.17%
gtar	11055	0.30	0.31	3.33%
gzip	724	1.55	1.59	2.6%

Table 3: Unix utilities sandboxed during the procedure of sandboxing RVM's installer script.

Unix Utilities/Commands			
File System	Privilege	Lookup and Search	Others
			tee
mkdir		Which	awk
mktemp		cat	head
rm	chmod	file	stat
date	chown	grep	sed
uname		find	xargs
cp		sort	df

be proxied to the supervisor process. The evaluations show that in the most time-consuming mode, the sandboxed version of RVM's installer, as our heaviest case of study, is 10% slowed down.

The focus of this study was on installer shell scripts and the corresponding utilities, which are not in the category of time-sensitive applications like real-time systems. Therefore, this performance is acceptable, considering that robust security is achieved as the supervisor executes the target program in the least privileged environment. In case of being compromised, the violating call will fail due to capability violations or failure in accessing the asked information, which ends up with process termination in the worst case, and no system corruption or information leakage would threaten the system.

5.2.1 Comparison with Unix Technologies

In this section, we compare Leash with similar lightweight application-level systems that are able to transparently sandbox one application in Unix. Table 4 shows the results obtained from isolating RVM's installer script using FreeBSD's Jail (FreeBSD Foundation, 2000), compared to Leash. Jail is a container-based isolation system for Unix that does not boot a new OS on the host OS but simulates the file system and its environment mainly based on the idea behind `chroot(2)`. In terms of the complexity of setup and configuration, Jail needs a container, an instance of the OS, to be made even for isolating just one application. It also requires root privilege for setups. In terms of the required system resources, jails occupy considerable disk space, as shown in Table 4. This can be a barrier to using them for trivial cases or when there is a lack of space. Starting a Jail service and executing an application inside it is much slower, 3.43 times in our test cases, than sandboxing by Leash. However, Jail cannot be applicable for some cases, such as updating or installing changes to the home directory.

We also compared Leash with CapExec (Jadidi et al., 2019). CapExec was a prototype that demonstrated limited transparent sandboxing on Unix. However, as it was tightly dependent on `libcasper(8)` in terms of the supported services, it was incapable of sandboxing many of the utilities and applications that are sandboxed in this study. CapExec was tested on small utilities such as `cat(8)` and `traceroute(8)` and demonstrated an average of 28% slowdown. It showed faster than Leash for smaller programs (fig. 3), but less capable and incomplete to be considered as a generic sandboxing system.

5.2.2 Comparison with Linux Technologies

As outlined in section 2.2, the implementation of "capabilities" in Linux differs significantly from true object capabilities. Consequently, there is no direct equivalent to Capsicum or Leash on Linux, though several sandboxing tools exist.

AppArmor, integrated into Ubuntu in 2007, is a Linux security model for defining Mandatory Access Control (MAC) rules(Gruenbacher and Arnold, 2007). Unlike Leash, AppArmor requires root privilege and static configuration. For application sandboxing, AppArmor is less flexible and more complicated to configure than Leash.

Decap, designed based on Linux "capabilities", is a tool for analyzing binaries to remove unnecessary privileges and enforce required "capabilities" by examining likely system calls (Hasan et al., 2022). It helps ensure that applications execute with only the

Table 4: Comparison on Leash and FreeBSD's Jail in sandboxing RVM's installer script.

Performance Details	Usage Details	Execution of RVM's installer (Native Program)	Execution of RVM's installer with isolation			
			Leash	FreeBSD's Jail		
				Starting jail	Installation by jexec	Stoping the jail
Memory Usage	Maximum Resident Set Size	12196	90324	7044	9924	6620
	Average Shared Memory Size	269	97	91	228	65
	Average Unshared Data Size	33	6	24	26	10
	Average Unshared Stack Size	128	106	128	122	128
	Page Reclaims	276994	262932	17260	325479	13236
	Page Faults	0	0	0	0	0
	Swaps	0	0	0	0	0
	Block Input Operations	0	0	0	0	0
	Block Output Operations	2122	1806	29	1747	5
	Messages Sent	15337	57209	204	7514	168
	Messages Received	7912	71388	2051	6803	7010
	Signals Received	0	0	1	0	0
	Voluntary Context Switches	16177	45514	799	9891	566
	Involuntary Context Switches	3	6	7	6	0
Time (second)		5.68	6.25	15.39		
Required Disk Space (Container)		0	0	4.0 G		

necessary privileges, reducing attack surfaces. Like Leash, Decap is based on mapping system calls to required Linux "capabilities".

Similar to Decap, LiCA (Sun et al., 2022) is another Linux "capability" framework that identifies excessive "capability" assignments by analyzing code flow through LLVM Intermediate Representation (LLVM IR). Unlike Decap, LiCA does not offer "capability" enforcement, and to employ LiCA's result, the source code should change accordingly. LiCA and Decap do not address the problem of generic transparent sandboxing, and neither system defines the isolation mode, the disallowed behaviours, and resource access policies.

6 RELATED WORK

In this section, we review the alternative approaches for secure software installation, along with examples of sandboxing systems and Unix security features that facilitate application-level sandboxing.

6.1 Unix Sandboxing Tools

The `chroot(2)` system call is a key feature for process isolation in Unix-like systems, restricting a process to a specified file system hierarchy. This laid the foundation for later sandboxing tools like OpenBSD's `unveil()` and FreeBSD's `Jail(8)`. While `unveil()` is similar to `chroot()`, it improves security by allowing developers to set specific permissions for the root directory.

RLBox is a C++ framework integrated with the Clang compiler, designed to facilitate the creation of a secure memory boundary for applications. It provides API support, type checking, and data marshaling to control data flow to and from sandboxed libraries (Narayan et al., 2021). One of RLBox's notable features is its API for cross-sandbox communication, allowing secure interactions between different sandboxed contexts. Central to this framework are "tainted types," which drive a type-safe approach; however, it requires modifications to existing code and emphasizes memory safety.

Other approaches to sandboxing include OpenBSD's Pledge and Linux's Seccomp-BPF, both of which impose restrictions based on a defined set of allowed system calls (Hughes, 2015; Linux Kernel Documentation, 2012). Developers can specify which system calls a target program can use, enhancing security by reducing potential attack surfaces. When a process calls pledge(), it enters a restricted environment where only the designated system calls are permissible. In contrast, while Seccomp-BPF provides a more complex and flexible mechanism for defining non-blocking system calls, it also allows for redefinition and replacement of calls, effectively acting as a static interposition mechanism. Despite its flexibility, both Pledge and Seccomp-BPF necessitate modifications to the application's code.

FreeBSD's Capsicum framework takes sandboxing a step further by establishing a robust sandbox mode that limits global namespaces, ensuring that these restrictions cannot be bypassed once `cap_enter()` is called (Watson et al., 2010; Anderson, 2017). Utilizing Capsicum requires significant code redesign, but it offers enhanced security by preventing unauthorized access to resources.

Additionally, FreeBSD's `libcasper(8)`, a library derived from Capsicum, provides a suite of widely-used services designed to operate within Capsicum's sandbox. These services include Capsicum-compatible APIs for networking, DNS resolution, file handling, and specific system calls like `getgrent(3)`, `getpwent(3)`, `sysctlbyname(3)`, and `syslog(3)`. While `libcasper(8)` simplifies the developer experience, it still confines users to a limited set of features.

Several sandboxing methodologies focus on tracking and interposing system calls. For instance, Li et al. proposed a model for intrusion detection that relies on system call interception, control, and data flow assessment (Li et al., 2010). Their system intercepts each system call, evaluating it against a predefined policy to ensure it aligns with expected behaviour. While effective in capturing system call contexts with manageable overhead, it faces challenges with policy complexity and performance for larger applications. Similarly, Ul Haq et al. described an isolation system for Linux that integrates Seccomp-BPF, AppArmor, Dune, and ptrace to restrict application access to system resources (Linux Kernel Documentation, 2012; Gruenbacher and Arnold, 2007; Belay et al., 2012). Their approach translates defined restrictions into Dune, leveraging hardware virtualization extensions (Intel VT-x) for a virtual machine-like isolation. The policy definition is complex, requiring extensive kernel interaction and high-level privileges, which increases performance overhead. In contrast, the approach adopted in Leash eliminates the need for hardware involvement. It simplifies policy definition using available resources, eliminating the need for complex translations or additional instructions. This framework prepares resources for the sandbox without requiring further verification.

Unix containers, such as FreeBSD's Jail(8), offer a form of sandboxing by creating isolated execution environments that mimic the host OS. While they provide good isolation, containers incur high memory and disk overhead, making them inefficient for sandboxing individual applications, especially when secure modifications to the host system are needed. Additionally, securing host machines against exploits originating from containers or hyper-visors remains a challenge as described in recent studies(Bhanumathi et al., 2023; Win et al., 2017). This ongoing debate highlights that, while containers offer isolation, they do not provide the same level of security as sandboxing, particularly in protecting the host from guest applications and vice versa.

In summary, sandboxing encompasses a range of defensive strategies designed to impose behavioural restrictions on programs, thereby mitigating risks and potential damages to systems from exploits. The primary objective is to prevent or minimize future harm and information leakage resulting from vulnerabilities in applications. Approaches addressing issues such as malicious code, application restrictions, confinement, and program compartmentalization all are counted in this category (Greamo and Ghosh, 2011). Designing a sandboxing mechanism targets specific vulnerabilities, defining limitations that prohibit certain behaviours through instructions, system calls, and access controls (Ansel et al., 2011; Watson et al., 2010). Variations in resource protection, behaviour definitions, and limitations lead to different sandboxing mechanisms with distinct purposes and effectiveness.

6.2 Secure Package Management

This study primarily was focused on enhancing the security of installer applications, which are critical components in software deployment. In this section, we delve into various security mechanisms that have been implemented in package managers on Linux and Unix systems. Package managers are crucial for verifying software integrity and auditing known vulnerabilities before installation, thus ensuring system security. Copper et al. provide an in-depth analysis of these security mechanisms (Cappos et al., 2008). While Unix and Linux package managers were not initially security-focused, they have evolved to address various concerns. For example, the Advanced Packaging Tool (APT) in Debian-based distributions uses Secure APT, which authenticates repositories with cryptographic signatures to ensure packages come from trusted sources. However, this mechanism lacks a confinement-based model, potentially limiting its effectiveness in certain scenarios.

FreeBSD's package manager includes key security features, such as auditing archives for known vulnerabilities, verifying package checksums, and issuing warnings about installation privileges (FreeBSD Foundation, 2023). The `pkg(8)` utility also operates within a sandbox using Capsicum, adding protection by isolating package management operations. However, these security measures are limited by the repositories supported and the vulnerabilities already identified. Poudriere is another package builder for FreeBSD that enables users to compile packages for multiple architectures on a single machine (Seaman, 2011). It enhances security by compiling packages within jails preventing unwanted remote connections. This approach ensures a clean build process, including only the necessary dependencies for the package's functionality. Nix, a non-native package manager, addresses the challenges of managing varying dependency versions (Kowalewski and Seeber, 2022). Originally designed for this purpose, it has evolved into a platform offering isolated, atomic, and reproducible package builds. Nix uses cryptographic hashes for each building entity, enabling users to detect any malicious alterations to source code before installation, thereby improving software management security.

In addition to package managers, several tools specialize in scanning and auditing software packages, such as *Package Analysis*, *NPM Audit*, GitLab's *Dependency Scanning*, and *Container Scanning* (GitHub Team, 2020; NPM Documentation, 2018; GitHub Team, 2018b; GitHub Team, 2018a). These scanners perform static analysis to validate software and its dependencies, identifying vulnerabilities and security risks. However, none of these studies have examined the dynamic analysis and monitoring of installation programs, which could offer valuable insights into real-time security during software installation.

7 FUTURE WORK

From a software engineering perspective, there are opportunities to enhance Leash into a general-purpose transparent sandboxing system on FreeBSD that can support a wide range of applications. This can be achieved by interposing the whole set of system calls prohibited under Capsicum's sandboxing mode. Currently, Leash supports applications whose disallowed functions are a subset of the list showed in Table 1, redirecting such calls to a supervisory module, except when they can be resolved based on the working directory.

In addition to supporting more functions, optimizing the policy specification could reduce communication overhead and improve system call handling, ultimately leading to better performance. Currently, Leash operates based on user-defined policies, which can become a usability bottleneck. While Leash's logging mode helps by providing a list of disallowed calls to restricted resources, user verification is still required. Our goal is to simplify this process and enhance user transparency through more readable and convenient policy specification. This area could be a focus of future improvements, where we could also explore the integration of AI technologies. In summary, by extending support for system calls, optimizing policies, and improving user interface clarity, Leash has the potential to become a robust, flexible, and user-friendly sandboxing solution for Unix.

8 CONCLUSION

In this paper, we described the design and behaviour of our prototype towards a general-purpose sandboxing supervisor for Unix. Leash is designed based on FreeBSD's capability-based security platform, Capsicum. To our best knowledge, this is the first transparent sandboxing supervisor designed for FreeBSD, which executes the target program in the sandbox mode without needing code modification and high access privileges such as root.

Compared to previously Capsicum-derived modules, such as `libcasper(8)` and CapExec (Jadidi et al., 2019), Leash provides new features and improvements. The system is a user-interactive framework designed for risk mitigation according to programs behaviours. Leash categorizes programs' behaviours into four primary domains: file system requests, network communications, system control requests, and process management.

We have evaluated the behaviour of Leash by sandboxing the installer script of Ruby Version Manager (RVM), and some utilities including Curl, Fetch, Gzip and Gtar, as real-world examples. During the sandboxing of RVM's installer, there were other Unix commands needed to be sandboxed inevitably. So, this investigation ended up with sandboxing 21 of other Unix commands, all described in section 5. Leash supports the interposition and functionality maintenance of the disallowed system calls listed in Table 1. Our evaluation shows that the sandboxed RVM's installer, as our most complicated case study, is 10% slower than the original version by average. This overhead contains managing 1009 forked processes, which had to run in the sandbox, too. In total, 17433 system calls were interposed, from which 12191 were proxied and managed by the supervisor process. This study shows that, despite the intensive restrictions that Capsicum applies, Leash makes transparent sandboxing achievable and provides practically efficient service to secure untrusted applications.

ACKNOWLEDGEMENTS

We gratefully acknowledge the support of the NSERC Discovery program (RGPIN-2015-06048).

REFERENCES

Anderson, J. (2017). A comparison of unix sandboxing techniques.

Anderson, J. P. (1972). The protection of information in computer systems. In *Proceedings of the 1972 ACM Annual Conference*.

Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B. (2011). Language-independent sandboxing of just-in-time compilation and self-modifying code. ACM.

Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. (2012). Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*.

Bhanumathi, M, K., S K, S. G., and Deb, P. (2023). Securing docker containers: Unraveling the challenges and solutions. In *2023 Fourth International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE)*.

Cappos, J., Samuel, J., Baker, S., and Hartman, J. H. (2008). *Package management security*. University of Arizona.

Dennis, J. B. and van Horn, E. (1975). Capability-based addressing. In *Proceedings of the ACM Annual Conference*.

Fabry, R. S. (1974). Capability-based addressing. ACM.

FreeBSD Foundation (2000). An introduction to freebsd jails. Accessed June 11, 2024.

FreeBSD Foundation (2023). Chapter 4. installing applications: Packages and ports. Accessed June 11, 2024.

GitHub Team (2018a). Container scanning documentation. Accessed June 11, 2024.

GitHub Team (2018b). Dependency scanning documentation. Accessed June 11, 2024.

GitHub Team (2020). Package analysis documentation. Accessed June 11, 2024.

Greamo, C. and Ghosh, A. (2011). Sandboxing and virtualization: Modern tools for combating malware.

Gruenbacher, A. and Arnold, S. (2007). Apparmor technical documentation. *SUSE Labs/Novell*.

Hasan, M. M., Ghavamnia, S., and Polychronakis, M. (2022). Decap: Deprivileging programs by reducing their capabilities. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, New York, NY, USA. Association for Computing Machinery.

Hughes, J. (2015). Pledge: Lighter than a jail. In *Proceedings of the 2015 USENIX Annual Technical Conference*. USENIX Association.

Jadidi, M. S., Zaborski, M., Kidney, B., and Anderson, J. (2019). Capexec: Towards transparently-sandboxed services. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–5. IEEE.

Kowalewski, M. and Seeber, P. (2022). Sustainable packaging of quantum chemistry software with the nix package manager.

Li, Z., Cai, H., Tian, J., and Chen, W. (2010). Application sandbox model based on system call context. In *2010 International Conference on Communications and Mobile Computing*.

Linux Kernel Documentation (2012). Seccomp bpf (secure computing with filters). Accessed June 11, 2024.

Linux man-pages project (2022). libcap. Accessed: 2024-12-1, Last edit: 2022.

Linux man-pages project (2024). Capabilities. Accessed: 2024-12-1, Last edit: 2024.

Narayan, S., Disselkoen, C., and Stefan, D. (2021). Tutorial: Sandboxing (unsafe) c code with rlbox. In *2021 IEEE Secure Development Conference (SecDev)*, pages 11–12. IEEE.

NPM Documentation (2018). Auditing package dependencies for security vulnerabilities. Accessed June 11, 2024.

RVM (2009). Rvm: Ruby version manager - rvm ruby version manager. Accessed on May 11, 2024.

Seaman, M. (2011). Poudriere - ports build and test system.

Sun, M., Song, Z., Ren, X., Wu, D., and Zhang, K. (2022). Lica: A fine-grained and path-sensitive linux capability analysis framework. New York, NY, USA. Association for Computing Machinery.

UCL Documentation (2014). Ucl - universal configuration language | github. Open-source project, Accessed June 11, 2024.

Watson, R. N., Anderson, J., Laurie, B., and Kennaway, K. (2010). Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium (USENIX Security 10)*.

Win, T. Y., Tso, F. P., Mair, Q., and Tianfield, H. (2017). Protect: Container process isolation using system call interception. In *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*.