Lightweight Cloud Application Sandboxing

Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi

Università degli Studi di Bergamo, Bergamo, Italy
name.surname@unibg.it

***Abstract*—Modern cloud applications can quickly grow to an elaborate and intricate tangle of services. In this scenario, paying attention to security aspects is important to mitigate the impact of incidents. Indeed, several research works and industrial standards recommend the integration of least privilege policies to prevent disruptions such as file system tampering. Unfortunately, technologies like containers virtualize file system resources with a volume-based approach, which may be overly coarse.**

In this work we address this problem proposing an approach that restrict application access to file system resources with a resource-based granularity. To this end, we develop a flexible and intuitive tool that relies on instrumentation to collect, merge, and audit the activity traces generated by any application component. We then demonstrate how this information is used to create fine-grained access policies, and introduce sandboxing using recent kernel security modules, strengthening the security boundary of the whole application. In the experimental evaluation we showcase the mitigation capabilities associated with our approach, and the low performance footprint. The proposal is associated with an open source implementation.

***Index Terms*—Cloud, monitoring, instrumentation, sandbox**

I. Introduction

Cloud applications are often built of many components, each implementing a specific set of business requirements. Components naturally evolve, their requirements may change, and as the overall scale of the application grows it can be challenging to ensure a high security standard. Many factors contribute to making this objective hard to achieve; the most common ones are: the presence of buggy components, the reliance on potentially vulnerable native libraries written with memory unsafe languages, and broad access to system resources.

Several research works have investigated the scenario (e.g., [1], [2], [3], [4], [5]). For instance, Staicu et al. [1] explain the security problems that arise when web applications interact with native extensions. BinWrap [2] and NatiSand [3] analyze recent vulnerabilities and integrate new security measures in the Node.js and Deno runtimes to improve isolation and limit access to confidential resources. Lastly, Zimmermann et al. [4] present an extensive study of third-party dependencies, finding that large parts of the entire web ecosystem can be impacted by security issues even when individual packages are vulnerable or they include malicious code on purpose.

Recently, industry standards have also emerged to encourage organizations to proactively include new security measures. A notable example is the NIST SP 800-190 [6], which focuses on environments that adopt microservices, containers and Kubernetes. The production environment is given particular attention, with the goal of finding and stopping malicious threats in real time. The directive clearly indicates that there is a need for policies to defend against vulnerabilities that could lead to disruptions, such as modification of important files. The same regulation also provides instructions to prevent the tampering of the file system, prompting that applications and containers must be run with a set of permissions as minimal as possible, namely following the *least privilege* principle.

Powered by the recent eBPF kernel technology, frameworks like Tetragon [7] and Falco [8] have been proposed to monitor cloud applications, identify unexpected security-relevant events, and act on them (e.g., denying access). While effective, they tend to introduce non-negligible overhead when fine-grained policy rules are used [9]. We argue that the combined use of classical operating system access control and sandboxing mechanisms may lead to a more resource efficient solution to isolate application components. For instance, the recent Landlock LSM [10] permits a process to restrict itself ensuring strong security guarantees with minimum performance footprint. Furthermore, the integration of Landlock, or an equivalent security mechanism, in cloud applications significantly mitigates the risk associated with the exploitation of a vulnerability, as the amount of resources available to a potential attacker is greatly reduced.

Unfortunately, to benefit from this protection developers must obtain a policy that clearly states the resources an application component must be granted access to, and the related permissions. This is far from trivial in the case of complex applications. Indeed, the list of resources can vary based on the production environment, or be subject to changes when different inputs are provided. All these reasons hold back the potential of sandboxing, and cloud applications may solely rely on the coarse isolation provided by the virtual machine or the container in which the application is executed.

Our contribution: In this paper we propose a new approach to systematically integrate fine-grained sandboxing in cloud applications that is aligned with the current regulations and best practices. In detail, we provide an intuitive, open-source solution to retrieve all the file system resources required by an application component, to build and customize least privilege policies. We then showcase how policies are used to sandbox programs with Landlock, mitigating the impact of severe CVEs. The approach we propose is flexible and does not depend on the toolchain leveraged to build each application component. The experiments showcase the minimal performance footprint at runtime, and highlight the ability to monitor the application without affecting its execution state.

The code is available at https://github.com/unibg-seclab/dmng

II. Motivation

This Section explains the threat model and the motivation.

A. Threat model

We assume that the code part of the cloud application (including native code executed by it) is trusted and not malicious, but potentially affected by vulnerabilities due to bugs. In this scenario, an attacker may leverage web interfaces or programmatic APIs to send the application malicious payloads with the goal of exploiting such vulnerabilities. This attack vector may leave the application exposed to, e.g., arbitrary file read and write, file system compromise, and execution of arbitrary programs; all of which can lead to an inconsistent state of the application and its volumes. We aim at the creation of fine-grained, component-specific policies that are used by developers to gradually introduce sandboxing, mitigating the impact of vulnerabilities. This approach is complementary to the use of containers, as a compromised process running in a container can still damage all the resources available to it.

B. Dependency identification

An important use case for cloud applications is represented by services that handle media resources such as videos, photos, and audio. These applications typically rely on an extensive set of editing libraries and codecs to perform a number of operations like crop, scale, introduction of effects, format conversion, and compression. This software is usually available as dynamic libraries that are loaded and executed depending on the type of operation to be performed, on the input source, and on the hardware support available. A solution able to automatically collect all the resources used by a component for a set of test cases can significantly help the developer detecting and isolating the dependencies of the application.

C. Mitigation of bugs

Open source libraries and programs are used extensively while developing cloud applications. Examples include database drivers, media processing libraries, and encoding utilities. This software is often trusted, but it may be subject to vulnerabilities as explained in Section II-A. When vulnerable third-party code is executed by the application, it can be targeted and compromised by an attacker who sends malicious payloads, as described in [11], [12], [13]. Popular cases are the Server-Side Request Forgery and Arbitrary File Read vulnerabilities found in FFmpeg and exploited against TikTok [14], and the Remote Command Execution vulnerability found in ExifTool and exploited against GitLab [15]. Other examples include: 1) CVE-2020-24020, CVE-2022-2566, CVE-2020-2499 associated with video processing software, 2) CVE-2022-1292, CVE-2022-2068, CVE-2022-2274 related to cryptographic software, and 3) CVE-2021-4118, CVE-2021-37678, CVE-2022-0845 targeting machine learning software.

With our approach we aim to support the progressive introduction of fine-grained policies that are used to sandbox the application or any of its components, making harder for an attacker to tamper the file system, corrupt data, or exfiltrate sensitive information such as private keys or database entries.

D. Performance and usability

Modern cloud applications are often packaged in containers and deployed in Kubernetes clusters as Pods. To improve their security and isolation Kubernetes provides a few integrated tools. For example, RBAC policies can be used to govern the behavior of software resources through *service accounts*. Unfortunately, these policies can only restrict access to Kubernetes APIs, therefore they are not suitable for fine-grained restriction of permissions at an application component level. The same limitation is shared with Seccomp profiles, which can limit the kernel interface available to the cloud application, but are only applied at container level [16] and cannot operate depending on the specific requested resource [17].

Recent solutions based on eBPF, like Tetragon and Falco, enable the introduction of fine-grained policy rules to overcome the previous limitations. To this end, they load into the kernel dedicated filters which are run system-wide every time a security event like the opening of a file or the execution of a program occurs. Based on the content of the policy, and therefore the filters, these frameworks can grant or deny a particular action, effectively restricting the privileges available to a cloud application. However, as mentioned in Falco's documentation [9], these approaches can introduce overhead subject to large variability. This is due to the evaluation of filters every time a certain hook point (e.g., a syscall) is triggered, and the need of multiple hooks to enforce a given security policy. Furthermore, these frameworks do not assist the developer in the generation of application-specific policies.

With our proposal we aim at giving the developer a complementary approach to secure applications and limit the file system resources accessible to them. Our idea is that the developer can benefit from the advantages brought by technologies like Landlock (i.e., strong security guarantees, low overhead), while at the same time rely on eBPF-based technologies, but only to monitor a restricted subset of important security events.

III. Approach overview

Frequently, developers start building cloud applications from base container images, which are subsequently customized and extended with third-party software (e.g., web frameworks, database drivers, etc.). Once the application has been developed, it is released to the staging area, a replica of the production environment, not accessible from the outside, where developers and cloud architects can test new features so to detect design flaws and prevent unexpected errors to hit the production environment. The similarity to the production area makes it the best candidate to generate accurate least privilege policies to restrict the permissions available to the application.

To operate as intended, each application component requires access to system resources like programs, scripts, dynamic libraries, shared memory, and files. We simply call these resources *requirements*. In order to collect them, we provide an intuitive open source tool called Dmng that performs service instrumentation as shown in Figure 1. In detail, the tool leverages ptrace and eBPF to setup and activate temporary probes that register the actions performed by an application

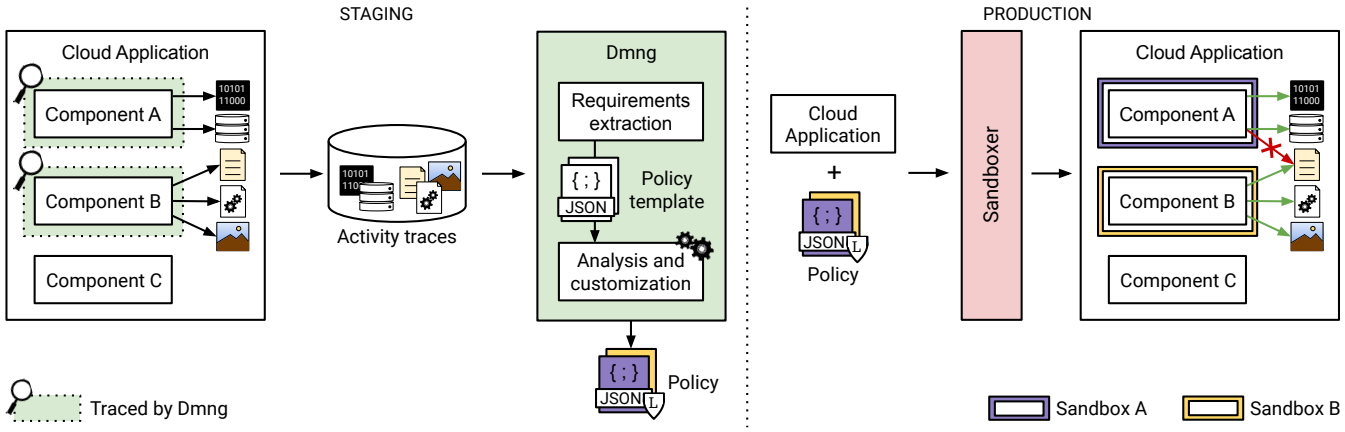


Fig. 1: Approach overview: 1) Dmng uses probes to trace the application components A and B, 2) activity traces are saved into a SQLite DB, 3) requirements are extracted from the traces and used to build security policies, finally 4) policies are leveraged by the sandboxer to secure the application in production

component, making it possible to track file-opening requests, reading and writing of data, use of shared memory and execution of native code via subprocesses and shared libraries. All these requests are automatically registered into a database and subsequently used to generate policy templates. Together with the file system path, each requirement is associated with permissions. Compatibly with Unix-like systems, three permissions are available: *read*, *write*, and *exec*. The policy templates generated by Dmng can then be interactively modified by the developer with the addition, modification or removal of policy rules. After the changes are committed, the policy is serialized into a JSON file and it is ready to be used to sandbox the application.

Sandboxing can be introduced with several technologies and frameworks; given that we aim to restrict file system resources, we provide a sandboxing utility based on Landlock that parses the set of requirements needed by the application (i.e., path and permission pairs) from the JSON policy file generated by Dmng, and it restricts the permissions accordingly. We selected Landlock due to its outstanding performance and stackability property. Indeed, the use of Landlock allows us to be compatible with systems that already rely on other LSMs (e.g., AppArmor, SELinux). We highlight that, whenever two or more LSMs are available on the host, a single denial prevents the access to a resource (i.e., deny takes precedence).

After the cloud application has been deployed to the production environment it is important to ensure that services are running as expected, there are no anomalies, and proactive measures are taken to identify potential threats. By default, access to any file system resource not listed in the policy is blocked by Landlock. However, there may be cases in which the developer would like to generate reports on the set of requested resources by the application. To enable this, Dmng allows to temporarily observe and record the activity traces generated by any application component without changes to the application itself nor its execution state. These checks are not bypassable, which is a considerable advantage compared to alternative techniques that either rely on LD_PRELOAD or perform instrumentation through code dependency injection.

The following sections are organized as follows. Section IV details the technologies used to support policy generation and implement monitoring. Section V clarifies the structure of the policy. Section VI describes sandboxing through Landlock. Lastly, Section VII showcases the mitigation capabilities and investigates the performance overhead.

IV. Cloud Application Instrumentation

Our solution implements two methods to collect the requirements and generate the policy. The first uses ptrace and is based on syscall argument inspection, the second leverages eBPF, which dynamically extends the kernel attaching dedicated probes on relevant in-kernel file system-related events. Both approaches are used to instrument the application, however, using ptrace significantly affects performance and thus is meant to be used only during staging. On the other hand, eBPF has lighter impact on performance, hence it can be used also in production.

A. Ptrace-based instrumentation

Ptrace is a functionality implemented by the kernel aimed at debuggers and code analysis tools that permits a process, called the *tracer*, to control and observe the activity performed by another process, the *tracee*. In our implementation, Dmng acts as the tracer for any application component. To this end, it prepares a parent and a child process as shown in Figure 2, and then uses the ptrace system call to instruct the kernel that the child will be traced by the parent (through the PTRACE_TRACEME request). After this step is completed, Dmng injects into the child process the component to be run, and then starts it. While being traced, every time an event occurs, the tracee is stopped by the kernel and a notification is sent to the tracer, which has the possibility to inspect and perform changes before the execution of the tracee is resumed. Dmng leverages ptrace to capture all file system-related syscalls, effectively monitoring the requests

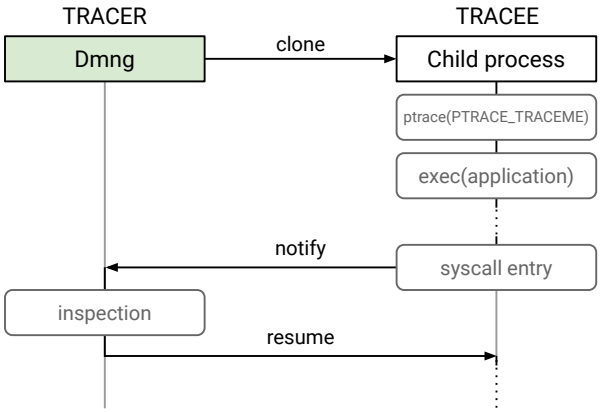


Fig. 2: Dmng acts tracer for the application, inspecting the arguments of every syscall

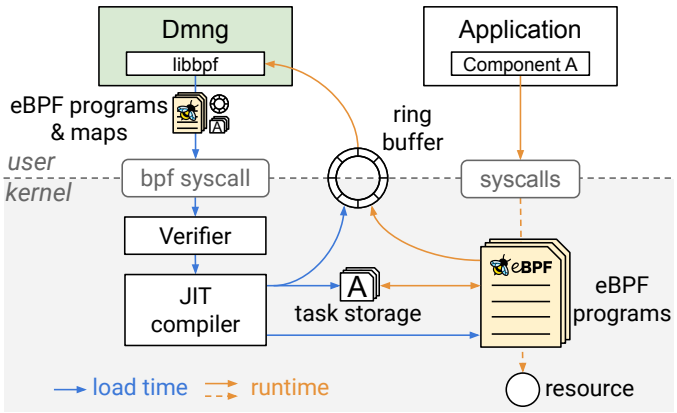


Fig. 3: Dmng uses libbpf to load the eBPF tracing programs and maps, then it polls data from the shared ring buffer

issued to the kernel by the component. The syscalls and their arguments are recorded by the tracer and saved to the SQLite database mentioned in Section III. The set of monitored syscalls includes interfaces such as open, openat, creat, execve, link, linkat, mkdir, and the related permission flags (e.g., O_APPEND, O_CREAT, O_RDONLY). It is important to point out that Dmng automatically captures and monitors the possible children spawned by the tracee, and it is also capable to identify the set of dynamic libraries they depend upon.

When the developer wants to stop the tracing process, Dmng detaches itself from the tracing of the child process with the PTRACE_DETACH request and it terminates the child process by sending a SIGKILL signal.

B. eBPF-based intrumentation

Similarly to other recent security and observability frameworks, like Cilium [18], and Tetragon [7], Dmng relies on the eBPF technology to implement continuous monitoring. The eBPF subsytem allows to change at runtime the behavior of the kernel without changing its implementation nor adding new modules. Briefly, it permits to do so by loading compact programs within the kernel, which are evaluated (without preemption) by a virtual machine-like component every time a certain hook point is reached. There are many types of hooks within the kernel, examples are network events, tracepoints, and LSM functions. To store data persistently between different eBPF program invocations and to share data between kernel and user space, data structures called *maps* are used. They provide abstractions such as arrays and hashmaps. It is important to mention that eBPF programs must be safe to run within the kernel and must not introduce bugs. To ensure these conditions are met, the eBPF susbsystem automatically performs the two stages of Program Verification and Just-In-Time Compilation at load time; only if both terminate without exceptions then the loading of the program is successful.

Dmng activates eBPF-based tracing on a given application component using its thread identifier. To this end, it leverages the *libbpf* [19] frontend to load into the kernel the eBPF programs and maps needed to perform tracing, and then starts collecting data. The process is shown in Figure 3. The set of eBPF programs comprises of: 1) dedicated programs to trace the application component lifetime, and 2) programs to monitor the file system-related events generated by the component. The former group of programs ensure that monitoring extends to tasks spawned through the clone system call by the component. Hence, they are attached to the *sched_process_fork* and *sched_process_exit* kernel tracepoints. Instead, the programs that record file system-related events are attached to hooks reported in Table I. Whenever one of these hooks is triggered, the attached program writes the requirement path and the related permission to a ring buffer shared with the Dmng user space process. This permits Dmng to poll data regularly, and then to save it in the already mentioned SQLite database (Section III).

We highlight that the collection of data using this method has minimal invasiveness: no changes must be introduced in the code of the application, nor it is necessary to restart it to setup the process. Indeed, when the developer wants to stop tracing, the eBPF programs and maps loaded by Dmng are automatically removed, leaving the system unmodified.

V. Policy

In this section we present the structure of the policy, explaining how it can be customized. We also discuss aspects such as coverage and effectiveness.

Policy structure: The policy obtained from Dmng is a JSON file structured as a list of objects as shown in Listing 1. For each of them, a field *policy_name* identifies the application component the policy applies to, while the sections *read*, *write* and *exec* are used to configure the related permissions. The structure of the policy is flexible, and for each object only the field *policy_name* is required. Since the policy implements a *default-deny* model, an object that does not list any section in its body has no runtime permission, hence the corresponding component cannot access any file system resource. Listing 1 shows an example in which a component called *filter* is granted execution access to the Awk program (together with its shared libraries) to process the read-only *users.csv* dataset.

TABLE I: List of file system traced hook points

Hook name
fentry/security_file_fcntl
fentry/security_file_ioctl
fentry/security_file_lock
fentry/security_file_mprotect
fentry/security_file_open
fentry/security_file_receive
fentry/security_file_set_owner
fentry/security_inode_getattr
fentry/security_path_chmod
fentry/security_path_chown
fentry/security_path_chroot
fentry/security_path_link
fentry/security_path_mkdir
fentry/security_path_mknod
fentry/security_path_rename
fentry/security_path_rmdir
fentry/security_path_symlink
fentry/security_path_truncate
fentry/security_path_unlink
lsm/bprm_check_security
lsm/mmap_file

Policy customization: Dmng provides a CLI interface that works simultaneously with multiple data sources to produce the list of requirements. By default, it implements the logic to automatically merge the requirements collected with ptrace and the eBPF programs. Moreover, it permits to customize the policy interactively by adding, changing and removing requirements. For instance, it allows to delete requirements based on the permission mask (e.g., `r_x`, `r--`), or change the permission associated with all the requirements that match a given path regex (e.g., `/usr/bin/libnet.*`).
A useful feature implemented by Dmng is *permission pruning*. This function takes advantage of the structure of the Directory Tree [20] to help the developer lower the number of requirements in the policy by reducing the granularity of permissions. The reduction in granularity is based on a *pruning goal* set by the developer that represents the desirable maximum number of policy rules associated with an application component. To implement this feature, Dmng first uses the policy template to build a trie (or prefix tree), then starts pruning its branches iteratively following a best effort approach, until the pruning goal is achieved. The rationale is that there are areas of the file system in which fine granularity brings strong security guarantees (e.g., `/lib`), but there are also many other areas where fewer rules make the policy more concise without affecting security (e.g., `/share`). So, it is important to consider contextual information about the current prefix path to guide the pruning process. Moreover, we want to comply with the *write xor execute* memory protection policy whereby every file may be either writable or executable, but not both. Thus, limiting the propagation of potentially insecure configurations (e.g., no dynamic library stored in `/lib` must be writable and executable by the application). After the pruning process terminates, the changes to the template are audited by the developer, and can be committed or discarded.

Listing 1: Example of JSON file with single policy

```
{
  "policies": [{
      "policy_name": "filter",
      "read": [
        "/lib/x86_64-linux-gnu/libsigsegv.so.2",
        "/lib/x86_64-linux-gnu/libreadline.so.8",
        "/lib/x86_64-linux-gnu/libmpfr.so.6",
        "/lib/x86_64-linux-gnu/libgmp.so.10",
        "/lib/x86_64-linux-gnu/libm.so.6",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib/x86_64-linux-gnu/libtinfo.so.6"',
        "/lib64/ld-linux-x86-64.so.2",
        "/usr/bin/awk",
        "users.csv"
      ],
      "exec": [
        "/lib/x86_64-linux-gnu/libsigsegv.so.2",
        "/lib/x86_64-linux-gnu/libreadline.so.8",
        "/lib/x86_64-linux-gnu/libmpfr.so.6",
        "/lib/x86_64-linux-gnu/libgmp.so.10",
        "/lib/x86_64-linux-gnu/libm.so.6",
        "/lib/x86_64-linux-gnu/libc.so.6",
        "/lib/x86_64-linux-gnu/libtinfo.so.6"',
        "/lib64/ld-linux-x86-64.so.2",
        "/usr/bin/awk"
      ]
  }]
}
```

Coverage and effectiveness: To generate the policy templates, Dmng registers the activity performed by the application while a set of test cases is executed. This approach is similar to the one proposed by the Slim toolkit [21] to identify the dependencies of a container and minify its image, and to the one followed by the Google Sandbox2 utility [22] to retrieve the requirements of programs distributed as ELF files. However, test-based policy generation can be subject to coverage issues if the set of test cases is not exhaustive. Another aspect worth mentioning is that poorly structured applications may benefit less from the isolation properties provided by sandboxing. Since Dmng supports the sandboxing of components with a per-thread policy, we recommend to leverage this function and execute potentially vulnerable components in dedicated compartments.

VI. Application Sandboxing

In this work we implement the sandbox leveraging Landlock, an unprivileged sandboxing mechanism officially merged into the Linux kernel in 2021 (version 5.13), with the goal of mitigating the security impact of bugs and unintended or malicious behavior in user-space application. The main reasons why Landlock was preferred to alternative sandboxing solutions such as Google Sandbox2 [22] are: 1) it does not rely on a proxy to implement the restrictions, hence it ensures low overhead at runtime, and 2) it is directly implemented within the kernel, thus it provides strong security guarantees. This section clarifies how policies are enforced with Landlock. Furthermore, it explains how `rwx` policy rules are translated into the Landlock permission model, and how restrictions are inherited by new components dynamically spawned at runtime.

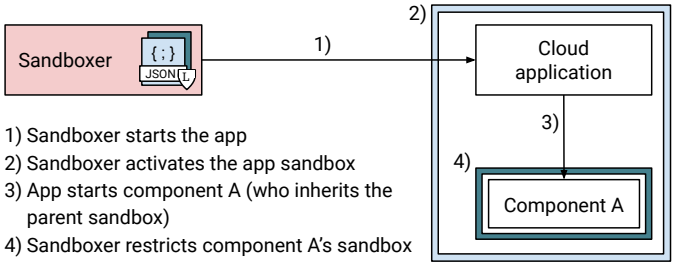


Fig. 4: Landlock sandbox setup and inheritance

The sandboxer is an extension of Dmng written in Rust that receives the JSON policy as input and modifies the application start procedure setting the permissions available to components before they are executed. The first task performed by the sandboxer is then to translate the `rwx` policy rules into the action-based permission model implemented by Landlock. In detail, Landlock groups permissions into *rulesets*, which collect the *actions* (e.g., FS_EXECUTE, FS_READ_FILE) permitted on each *object* (e.g., file, directory). The sandboxer separates the available actions to match the `rwx` categories, and then leverages the *landlock_create_ruleset()* and *landlock_add_rule()* interfaces to populate the rulesets accordingly. To activate the restrictions, a call to *landlock_restrict_self()* is perfomed. The process is illustrated in Figure 4.

An important property defined by Landlock is *policy inheritance*. Whenever a new component is dynamically spawned by the application in a child process, it automatically inherits the restrictions set on the parent. Moreover, after a ruleset has been activated, no new permissions can be granted to a component, as the ruleset can only be further restricted. Figure 4 shows the policy inheritance process for a generic application. The figure also shows how the inherited ruleset is further narrowed with a subsequent call to *landlock_restrict_self()* by the component.

VII. Experiments

This section presents our experimental evaluation. In the first part (Section VII-A), we reproduce a sample of CVEs affecting open source software, and showcase how the sandbox mitigates the exploits. In the second part (Section VII-B), we analyze the performance overhead. Specifically, we implement an application that performs various operations on media resources, and then evaluate the degradation of latency when sandboxing and eBPF-based monitoring are activated. The tests have been executed on a workstation with Arch Linux, kernel version 6.4, an AMD Ryzen 5 7600X CPU, 32 GB RAM, and 1 TB SSD.

A. Mitigation of vulnerabilities

To demonstrate the importance of the introduction of sandboxing, we selected the sample of high severity vulnerabilities reported in Table II. These vulnerabilities affect software extensively used in cloud application development such as FFmpeg, ImageMagick, OpenSSL and Exiftool. For each of them, we installed on the system a vulnerable version of the program or library, and then verified it was exploitable using

TABLE II: Sample of CVEs reproduced in our evaluation

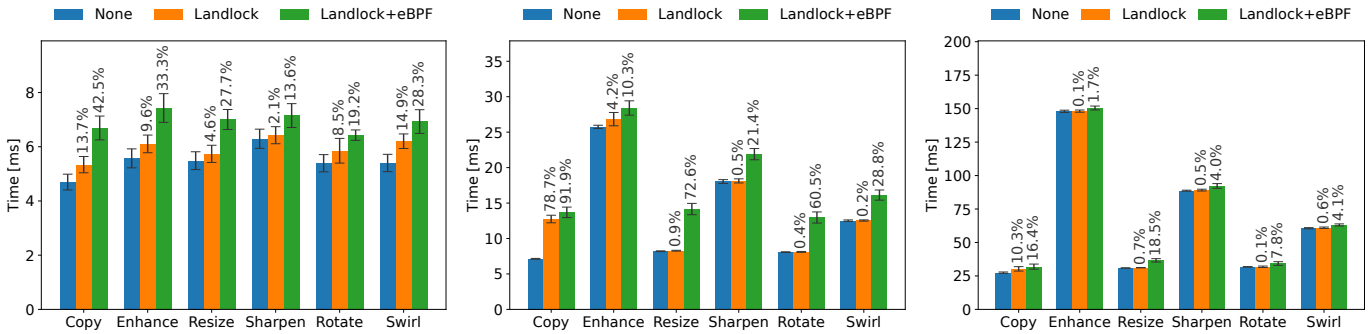
CVE	Software	Version
A crafted AVI video is used to read arbitrary files		
CVE-2016-1897 CVE-2016-1898	FFmpeg	v2.x
A bug in the PDF codec enables arbitrary code execution		
CVE-2020-29599	ImageMagick	v7.0.10-36
Improper sanitisation allows command injection		
CVE-2022-1292	OpenSSL	v3.0.2
Improper neutralization of user data in the DjVu file format is used to run arbitrary executables		
CVE-2021-22204	ExifTool	v12.23

public Proofs of Concept when the input is sent through programmatic APIs or web interfaces. Subsequently, we leveraged Dmng to generate least privilege policies. Finally, we repeated the execution of the previous tests starting each vulnerable component with our sandboxer. When benign input was submitted to the application, we were able to conduct the tests without loss of functionality. When instead malicious input was used, Landlock correctly blocked the exploit, limiting access to only resources listed in the policy. We highlight that, in general, similar protections extend to a broader set of CVEs.

B. Overhead

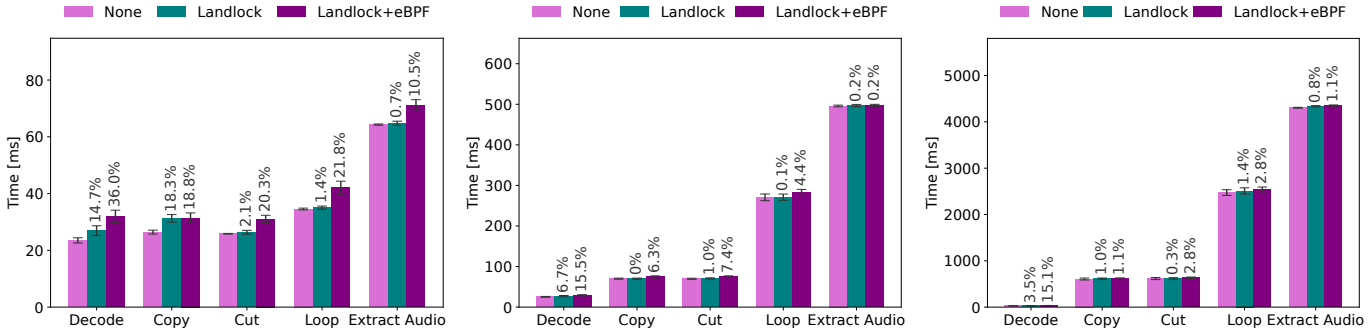
As mentioned in Section II, an important use case for cloud applications is represented by services that handle media resources such as videos, photos and audio. Therefore, to evaluate the overhead associated with our approach we implemented a Rust application that, upon receiving a request, leverages third-party software to apply several transformations on a media resource. Our goal is to measure the latency of the application to perform a given operation. Three test configurations are used: 1) no sandboxing is applied, hence no protection, 2) sandboxing is enabled leveraging our Landlock-based sandboxer, and 3) sandboxing is enabled plus eBPF-based continuous monitoring provided by Dmng is activated. Each operation is repeated 1000 times, and the measures are reported with 95% confidence intervals. Moreover, we focus on the server-side execution time, hence we do not consider the delay introduced by the network, which may make harder to visualize latency degradation for short-lived operations.

The first set of experiments focuses on image processing. In detail, the application leverages *convert* to copy, enhance, resize, sharpen, rotate and swirl images with 32x32, 640x480 and 1920x1080 resolutions. The results are shown in Figure 5. Inevitably, sandboxing introduces a slight degradation of latency compared to a scenario without protection. However, the overhead is non-negligible only for short-lived operations that last less than 10 ms, a duration that is considerably less than the average network delay. When instead eBPF-based monitoring is enabled, the data show worse latency degradation, especially for operations that last less than 50 ms. Remarkable is the case 640x480, in which the average operation overhead associated with eBPF-based monitoring is 47.6%.



(a) 32x32 image resolution (b) 640x480 image resolution (c) 1920x1080 image resolution

Fig. 5: Latency associated with various operations for an image processing application



(a) 6 seconds 480p video (b) 1 minute 480p video (c) 10 minutes 480p video

Fig. 6: Latency associated with various operations for a video processing application

The second set of experiments focuses on video processing. In detail, the application leverages *ffmpeg* to decode, copy, cut, loop and extract audio from videos with 480p resolution and with 6 seconds, 1 minute, and 10 minutes duration respectively. The results are shown in Figure 6. The overhead introduced by sandboxing is perceptible only for the decode and copy operations on the shortest video (6 seconds). However, it never exceeds 18.3%. As expected, the overhead becomes practically negligible for operations that take longer than 100 ms. The same considerations extend to the eBPF-based monitoring, which again confirms to be associated with more degradation compared to the Landlock-only solution.

VIII. Related Work

Several research works have highlighted the importance of sandboxing and isolation techniques in modern software [23], [24], [25], [26], [27], [28]. Indeed, sandboxing plays a key role in many systems and is integrated in software such as browsers (e.g., Chrome [29], Firefox [30]), service managers (e.g., Systemd [31]) and document viewers (e.g., Acrobat [32]).

With specific reference to the cloud scenario, many recent proposals have investigated the use of sandboxing to mitigate vulnerabilities [3], [12], [33], [11], [1], [2], [4], [13]. In *NatiSand* [3] and *Cage4Deno* [12] the authors modify the Deno runtime to control the permissions available to applications running native code. *BinWrap* [2] proposes similar measures to restrict the permissions available to Node.js native add-ons. *SandDriller* [11] describes an approach based on dynamic analysis for detecting sandbox escape vulnerabilities for Node.js applications. Zimmermann et al. [4] and Ferreira et al. [13] study the risks associated with vulnerable or malicious third-party dependencies and propose possible install (and update) time countermeausures. In general, all the previous proposals address the issues associated with a specific runtime ecosystem. Conversely, we aim to secure applications independently of their build toolchain or runtime.

Virtual machines and containers are two fundamental technologies in modern cloud architectures. Both permit to virtualize resources and execute applications in an isolated environment. Virtual machines ensure stronger security guarantees at the cost of higher resource utilization with respect to containers. The main reason is that applications executed in separate virtual machines have a distinct set of resources and do not share the same kernel [34], [35]. With specific reference to our scenario, both these technologies are associated with coarse granularity. Indeed, when working with them developers grant the application access to volumes rather than single resources. So, we provide a complementary approach to enable the introduction of fine-grained, per-resource access rules.

Modern industrial platforms like Cilium [36] and Falco [8] rely on eBPF as the primary means to enforce security

policies in cloud applications. Cilium provides networking, observability, and security functions for container workloads, while Falco implements a threat detection engine for clusters. These solutions can prove difficult to configure. Moreover, as already mentioned in the paper, the performance of eBPF-based solutions is associated with large variability when fine-grained rules are used [9]. Therefore, we propose to complement these solutions by assisting the developer in the generation of least privilege security policies and using recent sandboxing technologies like Landlock, to reduce the overhead and strengthen the security boundary of the application.

IX. Conclusions

The mitigation of security bugs and vulnerabilities that affect cloud applications is an important topic. In this work, we presented an approach to support the introduction of security policies to restrict the file system resources available to an application. To facilitate adoption, a central aspect in our proposal is the support to policy generation. We provide an open source tool to generate and customize least privilege policies leveraging the powerful, yet complex, ptrace and eBPF kernel technologies. Compared to virtualization technologies such as VMs and containers, which are associated with coarse granularity, we demonstrate that our proposal enables the introduction of fine-grained, per-resource access rules. The experiments showcase the capability of our approach to mitigate severe CVEs at the cost of limited overhead.

While currently the protection is limited to the file system, the isolation can be extended to other subsystems (e.g., the network). This is a promising line of research for future work.

Acknowledgments

This work was funded by the European Commission - Horizon Europe within the GLACIATION project (101070141), by the European Union - NextGenerationEU within the GRINS project (PE00000018), and by the Ministero dell'Università e della Ricerca - PRIN within the POLAR project.

References

[1] C.-A. Staicu, S. Rahaman, Á. Kiss, and M. Backes, "Bilingual Problems: Studying the Security Risks Incurred by Native Extensions in Scripting Languages," in *USENIX Security Symposium*, 2023.

[2] G. Christou, G. Ntousakis, E. Lahtinen, S. Ioannidis, V. P. Kemerlis, and N. Vasilakis, "BinWrap: Hybrid Protection Against Native Node.js Add-ons," in *ACM ASIA Conference on Computer and Communications Security*, 2023.

[3] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "NatiSand: Native Code Sandboxing for JavaScript Runtimes," in *Research in Attacks, Intrusions and Defenses*, 2023.

[4] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Smallworld with High Risks: A Study of Security Threats in the Npm Ecosystem," in *USENIX Security Symposium*, 2019.

[5] R. Jolak, T. Rosenstatter, M. Mohamad, K. Strandberg, B. Sangchoolie, N. Nowdehi, and R. Scandariato, "Conserve: A framework for the selection of techniques for monitoring containers security," *Journal of Systems and Software*, 2022.

[6] NIST, "Application Container Security Guide," 2023. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-190/final

[7] The Tetragon authors, "Tetragon," 2023. [Online]. Available: https://tetragon.cilium.io/

[8] The Falco Authors, "Falco," 2023. [Online]. Available: https://falco.org

[9] ——, "What is the performance overhead or resource utilization of Falco?" 2023. [Online]. Available: https://falco.org/about/faq/#what-is-the-performance-overhead-or-resource-utilization-of-falco

[10] M. Salaün, "Landlock: unprivileged access control," 2022. [Online]. Available: https://docs.kernel.org/userspace-api/landlock.html

[11] A. AlHamdan and C. Staicu, "SandDriller: A fully-automated approach for testing language-based JavaScript sandboxes," in *USENIX Security Symposium*, 2023.

[12] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses," in *ACM ASIA Conference on Computer and Communications Security*, 2023.

[13] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *International Conference on Software Engineering*, 2021.

[14] Hackerone, "External SSRF and Local File Read via video upload due to vulnerable FFmpeg HLS processing," 2021. [Online]. Available: https://hackerone.com/reports/1062888

[15] CVE Mitre, "Gitlab Exiftool vulnerability," 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22205

[16] The Kubernetes Authors, "Restrict a Container's Syscalls with seccomp," 2023. [Online]. Available: https://kubernetes.io/docs/tutorials/security/seccomp/

[17] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable system call security with ebpf," in *arXiv*, 2023.

[18] The Cilium Authors, "Cilium," 2023. [Online]. Available: https://cilium.io

[19] libbpf, "libbpf," 2023. [Online]. Available: https://libbpf.readthedocs.io/en/latest/index.html

[20] Linux man pages, "hier," 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/hier.7.html

[21] Slim.AI, "SlimToolKit," 2023. [Online]. Available: https://github.com/slimtoolkit/slim

[22] Google, "sandbox2," 2023. [Online]. Available: https://developers.google.com/code-sandboxing/sandbox2

[23] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *USENIX Annual Technical Conference*, 2013.

[24] C. Wright, C. Cowan, J. Morris, James, S. Smalley, and G. Kroah-Hartman, "Linux Security Module framework," in *Ottawa Linux Symposium*, 2002.

[25] A. Berman, V. Bourassa, and E. Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System," in *USENIX Annual Technical Conference*, 1995.

[26] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting Fine Grain Isolation in the Firefox Renderer," in *USENIX Security Symposium*, 2020.

[27] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi, "SEApp: Bringing Mandatory Access Control to Android Apps," in *USENIX Security Symposium*, 2021.

[28] M. Albanese, A. De Benedictis, D. D. de Macedo, and F. Messina, "Security and trust in cloud application life-cycle management," *Future Generation Computer Systems*, 2020.

[29] Chromium, "Sandbox," 2023. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/design/sandbox.md

[30] MozillaWiki, "Sandbox Architecture," 2023. [Online]. Available: https://wiki.mozilla.org/Security/Sandbox/Process_model

[31] Debian, "Service sandboxing," 2023. [Online]. Available: https://wiki.debian.org/ServiceSandboxing

[32] Adobe Inc., "Sandbox protections," 2023. [Online]. Available: https://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/sandboxprotections.html

[33] M. Abbadini, M. Beretta, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi, "POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes," in *ACM ASIA Conference on Computer and Communications Security*, 2023.

[34] V. Casola, A. De Benedictis, M. Rak, and U. Villano, "Security-by-design in multi-cloud applications: An optimization approach," *Information Sciences*, 2018.

[35] ——, "Monitoring data security in the cloud: A security sla-based approach," in *Security and Resilience in Intelligent Data-Centric Systems and Communication Networks*, 2018.

[36] The Cilium Authors, "Cilium," 2023. [Online]. Available: https://cilium.io