StemJail: Dynamic Role Compartmentalization

Mickaël Salaün
French Network and Information Security Agency
mickael.salaun@ssi.gouv.fr

Marion Daubignard
French Network and Information Security Agency
marion.daubignard@ssi.gouv.fr

Hervé Debar
Télécom SudParis
herve.debar@telecom-sudparis.eu

ABSTRACT

While users tend to indiscriminately use the same device to address every need, exfiltration of information becomes the end game of attackers. Average users need realistic and practical solutions to enable them to mitigate the consequences of a security breach in terms of data leakage. We present StemJail, an open-source security solution to isolate groups of processes pertaining to the same activity into an environment exposing only the relevant subset of user data. At the heart of our solution lies dynamic activity discovery, allowing seamless integration of StemJail into the user workflow. Our userland access control framework only relies on the ability of user to organize data in directories. Thus, it is easily configurable and requires very little user interaction once set up. Moreover, StemJail is designed to run without intrusive changes to the system and to be configured and used by any unprivileged user thanks to the Linux user namespaces.

Keywords

compartmentalization; dynamic policy; role; user activity; sandbox; Linux; namespaces

1. INTRODUCTION

Our daily use of computing devices leads us to indistinctively use them for both professional and personal activities, which is even seen as cost effective and promoted through the "Bring your own device" (BYOD) trend. As a result, each and every device we own contains all kinds of personal and professional information, with a wide range of levels of sensitivity. This information is in turn duplicated for storage purposes - and sometimes distant backup (e.g. in the cloud).

We tend to download and execute on a same platform applications originating from sources we do not trust uniformly. However, this enables attackers to indiscriminately access our data. Typically, some compromised multimedia software can very well upload professional content stored in the directory next to our music repository.

ASIA CCS'16, May 30–June 3, 2016, Xi'an, China.

ACM ISBN 978-1-4503-4233-9/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897845.2897912

Thus, additional security properties such as separation of duties seem desirable. Unfortunately, this imposes a significant burden on unqualified users. This notoriously complicated task relies on informed system management such as fine-tuning of access controls. It also requires sensitive privileges on systems that should not be granted to just anyone.

The problem of protecting information from exfiltration is not new at all, neither is the need for compartmentalization. Many sandboxing or virtualization techniques attempt to address this problem. Some solutions are integrated in the operating system kernel (e.g. SELinux) while the others emulate some components of the kernel in userspace [15, 20, 17]. Most kernel-based access control systems are dedicated to the administrator of the machine whereas the kernel emulation in userland is a workaround to allow unprivileged users to compartmentalize their processes.

Rather than extending global system security, we advocate a new approach of user-oriented separation of activities. We point out that within the context of a given user, every executed application can access every piece of data owned by this user. Therefore, if information in itself is the sensitive target, an adversary does not need to perform a privilege escalation anymore. Assuming a perfectly well-managed and secure operating system, we want to provide a practical solution to allow users to perform their activities in a safe way. Namely, an attack should only result in leak or loss of data related to the activity actually executing the compromised software instance. An activity will then reflect a security policy formalized with a set of access rules, called user domain. To implement an unprivileged and non-invasive solution, we choose a hybrid architecture: a userland engine is used to define and dynamically update a policy, dedicated to a user, but enforced by the kernel.

In this paper, we present StemJail [23], a new practical and open-source security framework that can be applied to an existing GNU/Linux operating system without major modifications. StemJail allows *automatically inferring the user activity* according to his actions. This is the major contribution of our solution: dynamically mapping a security policy according to the user workflow enables to create a seamless dynamic access control without bothering the user with confirmation interruptions (e.g. modal window). Easily picked up by average users, configurable and usable without any administrative rights, StemJail provides users with a way to augment the security of their GNU/Linux systems with a compartmentalization guarantee which they can understand. Exhibiting a low performance overhead and

compatible with existing applications, we believe that our implementation constitutes a relevant complementary access control in addition to classical system-level measures.

Section 2 provides background on some relevant compartmentalization and sandboxing solutions including Janus [15], Ostia [14] and Mbox [17], before presenting our running example and requirements imposed on the project. Then, Section 3 explains automatic discovery of user activity. Section 4 details the architecture choices on which our implementation relies. The security guarantee provided is expressed and proved in a formal model in Section 5. In Section 6, we illustrate the workings of StemJail on our running example and benchmark its performances. Related work, along with limitations and future work are covered in Section 7.

2. COMPARTMENTALIZATION OF ACTIVITIES

2.1 State of the Art

Despite different underlying principles, existing virtualization and compartmentalization solutions share the same purpose: running multiple autonomous systems in isolated environments while sharing the same hardware. The fundamental model is based on a monitor managing subjects (i.e. kernels or processes). We categorize these isolation solutions in three main categories: hypervisor-based, kernel-based or userland-based, ordered by decreasing order of adherence to hardware. Hardware access level usually impacts the granularity of the control enforced by the monitor. For example, hypervisors alone are not able to enforce a fine grained file system access control because they work with raw devices or memory. The OS kernel is able to make sense of data stored, which will then be transmitted to processes. In addition, heavy hardware access requirements increase the invasiveness of a solution.

With respect to hypervisor-level monitors, Xen is a hypervisor used by security solutions such as Qubes OS [22], a single end user GNU/Linux/Xen distribution that provides tools to manage virtual machines (VM/guest) dedicated to a unique activity. The hypervisor is only a part of the compartmentalization mechanism of Qubes OS. Every guest communicates with the monitor, implemented by the hypervisor and an administrator VM (*dom0*). Moreover, the end user is also the administrator of the machine.

With respect to kernel-based monitors, Linux-oriented compartmentalization solutions include VServer, OpenVZ, LXC or Docker. They provide tools to create and manage light virtualization environments called jails or containers. These environments use the same kernel which can share resources grouped by namespaces (network, file system...). Each environment is created by the administrator.

Compared by Schreuders et al. [24], Linux-oriented access-control mechanisms such as SELinux, AppArmor, Tomoyo, Smack or grsecurity form another kind of kernel-based monitors. These solutions provide Domain Type Enforcement (DTE [3, 25]) and Mandatory Access Control (MAC) engines to create and manage fine-grained access policies to restrict processes. These policies complement the traditional Discretionary Access Control (DAC); again, they require administrative access for management purposes.

Newer operating systems, including Android and iOS, provide control of the applications to the end user. Those require some administrative rights but not the full control over the device. The permissions are high-level and meaningful to the user. However, he cannot express policies to control file sharing between applications. An orthogonal approach is the User Account Control (UAC [21]) used in Windows, which interactively requests the administrator to grant necessary permissions to an application.

There are many userland-based monitors and sandboxes [24], most using syscall interposition mechanisms, for example Janus [15] or Systrace [20]. More recent projects such as Minijail, Subuser, Firejail or Oz (Subgraph) create a dedicated confined environment to run applications. MAPbox [1] looks at the MIME-type of files to infer a configuration. Some, such as Alcatraz [18] or Mbox [17], interact with the user to validate file changes. Others use a static security policy. Ostia [14] uses a delegating architecture to outsource and emulate sensitive syscalls to a trusted process instead of the traditional filtering architecture. This kind of sandboxing helps the end user to enforce access rules for unmodified applications but require administrative privileges (e.g. SUID binary) or induce a significant performance penalty.

Developer-oriented sandboxes, such as Capsicum [26] or Apple XNU Sandbox [8] (previously Seatbelt), and attack surface reduction mechanisms such as Seccomp-BPF [10], focus on privilege reduction for applications (e.g. web browsers). These sandboxes do not require administrative privileges but their security policies are hard-coded in the software.

2.2 Running Example

Many use cases illustrate the need for compartmentalization of data. While the traditional example is the separation between professional and personal use of the same underlying hardware (i.e. BYOD), we choose to present a less clichéd but potentially more complex use case as a running example in this paper. Namely, we choose to study the needs of a consultant adapting a proprietary piece of software for several customers.

Our consultant has to deal with two customer firms, `OpenBar` and `Paranoid`, which happen to have very different security expectations. Both companies entrust the consultant with their professional data, but security practices enforced at `Paranoid` outperform those in place at `OpenBar`. The idea is that, whereas it is very likely that `OpenBar` gets compromised, it should not result in the leak of data related to `Paranoid`. In other words, the software developed for `Paranoid` should not be retrievable by a malware originating from `OpenBar`. On his machine, the consultant is using generic applications, including a PDF reader that we call `Viewer`. In addition to working directly for clients, our user also carries out certain tasks only relevant to `Company`, the consulting firm to which he belongs.

Two distinct activities naturally emerge from the simple but realistic example described above. Everything that the consultant needs to work on the `OpenBar` project (resp. on the `Paranoid` project, or for `Company`) belongs to an activity which we call `OpenBar` (resp. `Paranoid`, or `Company`). Any relevant compartmentalization solution should provide the consultant with a way to perform both activities simultaneously: at a minimum, smooth task switch has to be allowed, e.g. to answer emails or phone calls.

While involving users with a certain level of computer literacy, our scenario is really not involving computer security experts. Hence, the separation between activities introduced

here corresponds to a natural one for the user, one with which he can come up by himself. Indeed, we do not assume our user to foresee things like the complexity and buggy nature of a viewer, or the natural vulnerability of an application dealing with untrusted data: we do not aim at sandboxing dangerous applications. We only assume as a requirement that the user of a confinement solution should be able to specify legitimate circles for circulation of information.

The problem of deciding to which circle a piece of data relates is tough to solve (e.g. the files MIME-type [1] is a partial solution). On the one hand, the user should not be burdened with troublesome and error-prone data tagging. On the other hand, we do not aim to implement some involved automatic inference procedure. We find a middle ground in the fact that a user can put to good use a confinement solution which he understands. Concretely, organizing data in directories fitting activities seems simple enough to bootstrap, manage and keep consistent over time. In our example, we assume that our consultant has created a directory `~/Clients/OpenBar/` (resp. `~/Clients/Paranoid/`, and `~/Company/`) where he stores files related to `OpenBar` (resp. `Paranoid`, or `Company`). Our hypothesis is that data in each directory is related to the activity giving its name to the directory.

Besides, our solution addresses classical problems rising from compartmentalization of execution environments, such as the ability to coherently list files pertaining to a given environment or to safely access common resources.

2.3 Functional and Security Requirements

When the user is the source of information, he is the one who knows the best what kind of classification or sensitivity the data should be marked with. With StemJail, empowering the user with the definition of the security policy does not put at risk the system security because the policies can only add more constraints and are only applied to the processes of this same user. Following our example, the consultant working on a project for a client can easily tell if his work can be shared with other partners or should be kept private.

Security policies require clearly stating which parts of a system are trusted, leaving the rest under the possible control of an adversary. In our case, the Trusted Computing Base (TCB) includes the Operating System (OS) kernel, the system services and the hardware. Moreover, attacks on side or covert channels are outside the scope of this paper. StemJail aims to provide users with a way of controlling information flow amongst their data, by means of compartmentalization of user activities. Therefore, StemJail supplies a framework enabling users to ensure the confidentiality and integrity of activity-specific data w.r.t. other activities. In other words, confining an activity allows us to mitigate data leaks in case the activity is compromised by adversaries abiding by hypotheses defined above.

On top of its core security goal, there are a few other extern functional and security constraints which StemJail aims to satisfy:

- To be effective, an access control framework dedicated to the end user should not bother him with interruptions. By being *fully integrated into the user's workflow*, StemJail is seamlessly enforcing access control.
- To fit the needs of each and every user of a system, StemJail needs to be *accessible and configurable by any of its users*, without administrative rights. Each user should configure the solution for use within his user sessions.
- In order to use it properly, users should understand the confinement underlying the access control solution which they configure. As a result, they should only have to *influence access controls in a manner which has concrete meaning to them.*
- Adding a new security feature such as StemJail should *not degrade the security level of the system.* In particular, we choose to rule out any modification of the TCB (e.g. of the kernel), so as to avoid introducing new critical security vulnerabilities. Moreover, access control features implemented by StemJail are not meant to replace system-wide security policies configured by system administrators. Security features brought by StemJail should complement those already in place on a well-managed system.
- *Compatibility with currently used applications* without requiring their modification really increases the practicality of a solution. We choose to address this challenge on current GNU/Linux systems. A Linux distribution can provide additional packages, if they do not alter the system too heavily. Hence, we impose on StemJail not to modify existing system components.
- Eventually, a security solution is expected to have some impact on performances. However, to be usable in practice, a solution should be as *efficient* as possible.

3. OVERVIEW OF STEMJAIL

3.1 Definitions

StemJail articulates its security policy around the idea of user activities. An *activity* is in essence quite an informal notion: it is a set of tasks coherent from the user's point of view. In this section, we introduce definitions formalizing the access control policy enforced in StemJail.

Role.

As introduced by Ferraiolo and Kuhn [11], a *role* refers to a job function which yields authority and legitimacy in performing related tasks. The need for isolation stems from the fact that users often assume multiple roles. The concept of user activity introduced above is naturally formalized as a role defined by the user. In our example, the consultant has a dedicated role when he works for a specific client (i.e. role `OpenBar` or role `Paranoid`). More generally, as some accesses can be legitimate for multiple activities, it is possible that the user assumes any disjunction of user-defined roles. We call these latter roles *intermediate*. For instance, the intermediate role "`OpenBar` *or* `Paranoid`" consists in tasks indifferently performed with role `OpenBar` or role `Paranoid`.

Object.

Following the RBAC [11] vocabulary, we name *object* the target of our access control policy, which is user data. Namely, in the scope of this paper, an object is what is commonly called a (filesystem) path. It identifies a directory (and its content) or a file of the filesystem on which StemJail is executed. As a result, the data referenced by a given object can appear, evolve or disappear over time. In our example, we assume that our consultant has created a directory `~/Clients/OpenBar/` (resp. `~/Clients/Paranoid/`) where he stores files related to `OpenBar` (resp. `Paranoid`).

Action and Access.

Actions are labels capturing operations on data referenced by objects. We only introduce in this paper two actions, `read` and `write`, obviously capturing read and write operations on underlying resources, but our model could be extended further. We define an *access* by an action performed on an object.

Rule.

A *rule* is a triple consisting in a role, an object and an action. Informally, when a user specifies a rule (r, o, a), he describes that he should legitimately be able to perform action a on object o when assuming role r. Given elements introduced above, we expect that our consultant needs to write on data stored in `~/Clients/OpenBar/` when he is working on the `OpenBar` project. This is described by the rule (`OpenBar`, `~/Clients/OpenBar/`, `write`).

Domain.

A *domain* is a set of rules. Like roles, domains fall into two categories: user and intermediate domains.

User domains are defined by users and consist of rules which share the same role component. We have said that a user activity is formalized as a role from the user's point of view. For each such role, the user must list the rules describing rightful actions on objects when carrying out the activity captured by the role. Each of these lists brings forth a dedicated configuration file defining a domain. We highlight here that this is one of the reasons why it is tremendously important that users of StemJail understand the isolation which they want to instore.

Intermediate domains are formed as intersections of *user domains*. Intermediate domains are the counterpart of intermediate roles: intuitively, if a rule appears in two user domains, a user conforms to this rule when he is assuming any one of the roles corresponding to the user domains.

Subjects and Monitors.

When an instance of StemJail is spawned, it creates a reference process called the *monitor*, while all other processes jailed in the new execution environment are called *subjects*. The monitor is in charge of controlling actions performed by subjects so that they conform to a domain. There is one monitor per instance of StemJail, and all instances of StemJail execute on behalf of the same user as far as the operating system is concerned. Architectural details are further discussed in Section 4.3.

3.2 Automated Role Discovery

The user can explicitly set the role he believes he will perform but it is error prone, cognitively burdening and not user friendly. We hence design a way to automate role discovery.

The monitor uses a state to store a domain, called the current domain, corresponding to the access control currently enforced. Intuitively, the idea is to start a subject execution with an empty set of accessible objects, and proceed by augmenting the set of possible accesses on the fly. The monitor must thus have a strategy to decide whether an access request from a subject should be granted. This strategy must guarantee that all legitimate sequences of access requests of subjects matching at least one of the user domains are granted. It must also guarantee that *only* such requests are.

When a subject starts its execution, three cases can arise. Its first access request on an object can appear in no user domain, only one of them, or several user domains. In the first case, it is easy to say that the request should be denied. The second case is simple enough too: it allows the monitor to determine at once to which user domain the activity corresponds. The monitor state, i.e. the current domain, should be set to this user domain, and all subsequent requests should conform to this particular domain specification. The last case is possibly the most frequent, but is a little more complex. The request should be granted, but the monitor state cannot transition to a given user domain.

This is where intermediate domains, which consist in actions on objects appearing in multiple user domains, come into play. Here, the monitor should set its current domain to the intermediate domain corresponding to the intersection of all user domains containing the requested object access. By doing so, the monitor captures that the activity is necessarily amongst those corresponding to these user domains. During the rest of the subject execution, as new object accesses are granted, the current domain should only be able to evolve to one of these user domains. Otherwise, no user domain would authorize the set of accesses to objects performed by the subject, while this is supposed to be our security guarantee.

Quite intuitively, the monitor strategy outlined here is implemented as an automaton on intermediate domains. A transition between a source and target intermediate domains exists if the target domain contains the source domain. This formalizes that all accesses deemed legitimate in the source domain remain so in the target one.

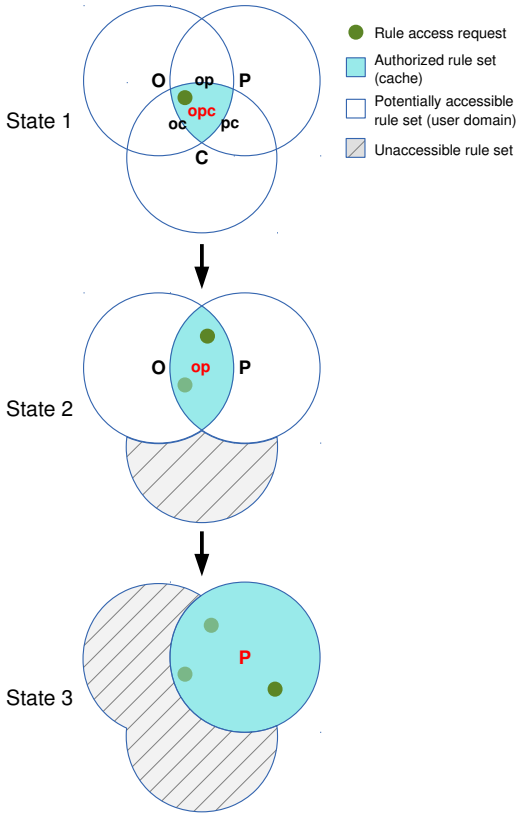

Figure 1: Domain Specialization by Transition

We illustrate a possible sequence of domain transitions in Figure 1. As chosen in Section 2.2, there are three user

domains: `OpenBar` (`O`), `Paranoid` (`P`) and `Company` (`C`). Some objects and accesses appear in rules of several domains. They are depicted as overlaps forming intermediate domains: `op`, `oc`, `pc` and `opc`. Figure 2 shows all the possible transitions. The scenario workflow is as follows.

1. A subject initially requests access to an object common to the three domains. The green dot symbolizes the corresponding rule in the intermediate domain formed by the intersection of the three domains: `opc` (`OpenBar` *or* `Paranoid` *or* `Company`). At this stage, all intermediate domains are potentially reachable, which means that all their files are potentially accessible by the subject.

2. When granting access to an object thanks to rules in domains `Paranoid` and `OpenBar`, the monitor transitions from `opc` to `op` (`Paranoid` *or* `OpenBar`). This transition is allowed because the origin domain (`opc`) is a subset of the destination domain (`op`). The current domain then describes a new set of rules, which enable the subject to perform the requested object access. Obviously, the subject can cache its requests and only query the monitor when a transition is required (cf. Section 4.4). The monitor state captures that the current role is not related to the `Company` domain, and no object access authorized only by this latter will be added by the monitor.

3. When a subject requests an access to an object described by a rule in `OpenBar` but not in `op`, the monitor transitions the current domain to `OpenBar`, which is a user domain. The previous accesses are still allowed but no more transitions are accessible.

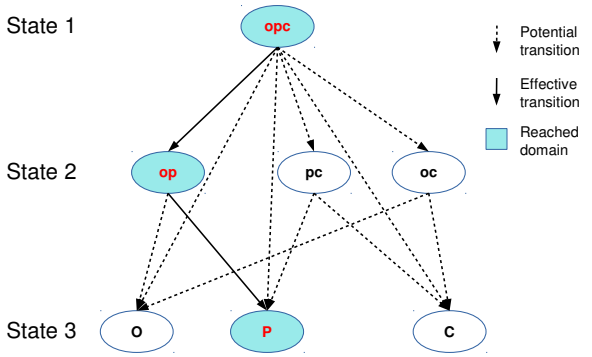

Figure 2: Lattice of Domain Transitions

We underline that role discovery does not require all user domains to have a common intersection. When an access request does not match any rule from accessible domains the request is denied. The monitor remains in the current domain and the subject will not be able to access the object.

Unlike the Bell and LaPadula's [5] model which reflects the military restrictions to prevent data leak, the Chinese wall [9] security policy describes commercial constraints to protect against conflicts of interest. Our approach is a lattice-based policy comparable to the Chinese wall model, but in a less compelling way. Indeed, using well-chosen user domains, StemJail can be used to implement a Chinese wall policy.

If a subject is compromised while its monitor can transition to multiple domains (e.g. from `opc`), all resources featured in the reachable domains could be compromised too. Therefore, to benefit the most from StemJail security guarantees, configurations should conform to the following guidelines. For each user domain, questionable data should be quarantined in directories dedicated to the domain before clean-up. Moreover, the data shared between multiple domains should be trusted and accessible in read-only mode to avoid the spread of malicious documents. The policy can then prevent a threat to propagate from a quarantine zone to other domain via shared read-write resources.

The worst case scenario of intermediate domains is $2^n - 1$ for n user domains. In practice, this exponential spatial complexity is not a problem for the monitor because of the small memory footprint a domain requires and the limited number of (almost similar) domains a user would need.

Automated role discovery deduces the user role based on his subjects' actions. Moreover, the file layout hierarchy is already understood without additional learning.

4. IMPLEMENTATION OF STEMJAIL

4.1 Architecture Overview

The architecture of StemJail is designed to dynamically adapt to multiple user activities, each of them being confined in jails. As shown in Figure 3, there are three main components: a portal, one monitor per jail and one or more subjects per jail.

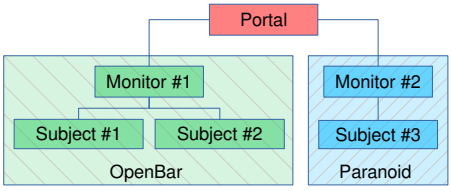

Figure 3: Overview of a StemJail Instantiation

The portal is responsible to initiate a jail creation by spawning a monitor and maintaining a communication channel to send commands and receive information.

Every monitor is the initial process of a jail and is hence dedicated to this jail. A jail's monitor is the only privileged process for this environment: it is the only one able to extend the jail with more accesses according to its policy.

Subjects are instances of user applications for one activity. Each subject in a jail is a client able to request new accesses to the monitor. This is its only way to gain new accesses, since it is unprivileged and remains so. Indeed, we rely on the kernel, part of the TCB, to prevent privilege escalation. Further internal details will be described in Section 4.3.

In this example, the `OpenBar` jail contains two subjects able to send requests to their monitor which currently follows the `OpenBar` security policy. These three processes can communicate between themselves but are isolated from the `Paranoid` jail and its subjects. This second jail can have private data protected from the view of the first jail.

Thanks to this architecture, StemJail is able to spawn as many jails as needed while still being able to manage them consistently.

4.2 Creating an Ecosystem for Subjects

As explained in Section 2.3, StemJail needs to be able to enforce visibility restrictions on data but it should neither

modify the kernel nor require privileges. To do so, StemJail relies on the Linux namespaces [6].

There are six namespaces available from the Linux kernel: mount, UTS, IPC, PID, network and user. Each one exposes an isolated custom subset of the kernel features to userland. They can be used to isolate processes from one another.

The mount namespace allows creating isolated filesystem mount points only visible to userland processes in that dedicated namespace. We can thus limit the filesystem view of some processes.

The user namespace is the only namespace available to unprivileged users. This allows users to change their ID in a dedicated environment and hence become a *root* user in this namespace. This user can only impact processes in the same or nested namespaces. While still constrained by the privileges granted to the original process (UID, GID), a *root* process in a user namespace is able to do some restricted administrative tasks like creating other kinds of namespaces.

In order to avoid the complexity and pitfalls of creating a new non-root compartmentalization system from scratch, StemJail relies on user namespaces. They are used to create custom jails. The *root* user of a user namespace can create a dedicated mount namespace. It is then possible to use bind mounts inside this namespace to expose a suitable subset of files and directories with read-only or read-write permissions.

A recursive mount point should not be created without looking at the inherited mount points. Since the mount options for a bind mount only apply to the first one, StemJail needs to handle nested mount points to avoid illegitimately exposing files. Besides, because of the user namespace restrictions, it is not possible to only mount a part of the file hierarchy. To avoid this default behavior, StemJail recursively mounts paths which should legitimately be exposed. The number of mount points can thus be large because of the numerous mount points in the host (i.e. more than 20 in a common GNU/Linux distribution). For any mount point and exposed file or directory, StemJail must first create the mount point, then remount it in read-only mode if needed, and then do the same for all its nested mount points. To avoid race-conditions, these adjustments are first performed in a private temporary directory before atomic exposition of resources to subjects. The exponential growth in bind mounts is manageable in practice because a mount point does have a small memory footprint.

Another important aspect of the mount management is to stay in control of the mount hierarchy. Assuming `/a` is bind mounted to `/b`, the problem is to control whether new mount points in `/a` should be propagated to `/b`. To manage this behavior, a mount point can be marked as shared, slave or private. If we did not forbid new mount point propagation, it would be trivial to bypass the policy, e.g. by mounting an USB mass storage to `/media` while this directory is exposed in read-only domains.

Most applications require access to specific Linux devices to run correctly. For example `/dev/null` and `/dev/urandom` are widely used. Devices are kernel interfaces which access potentially sensitive data (e.g. storage devices). Thus, a user namespace is not allowed to create devices. For this reason, StemJail bind mounts parent devices into every new domain. In a defense in depth approach, we choose to only expose a minimal set of devices. Therefore, we limit them to the strict minimum of 4 devices (`null`, `full`, `zero` and `urandom`).

The temporary directory `/tmp` is widely used by many applications. This directory is commonly shared between all users of the machine to store temporary files. To avoid any risk of misusing temporary files that can help an attacker to gain more privileges or access unexpected data, we choose to create one dedicated temporary directory per domain. It is much safer and still usable by applications. Moreover, this private directory can be used to store data while being sure other domains can't access them (e.g. a UNIX socket).

Subjects often need to list all the processes in their environment, but this should not enable them to interfere with processes executing in other jails. This requires exposing a *proc* filesystem in `/proc`. Thanks to the PID namespace, StemJail creates a pseudo-filesystem dedicated to each jail.

4.3 StemJail's Internals

The isolation provided by jails relies on Linux namespaces. Jails are separated from one another and from the system using the available namespaces: PID, network (if activated), IPC and UTS. Each process in a jail is therefore unable to see or interact with processes outside its jail. The filesystem view can however be shared in accordance with user-specified domains.

The communication channels between the subjects and the monitor are UNIX sockets. They provide trusted peer identification (PID, UID and GID) by means of `SCM_CREDENTIALS` messages, on which the monitor can rely. Moreover, sockets are opened in the `/tmp` directory, which is private and dedicated to the jail (cf. Section 4.2). This rules out attacks based on techniques such as the confused deputy scenario [16].

A filesystem attached to a jail is built incrementally. When a jail starts, the execution flow is as follows:

1. At first, the monitor (PID 1 in its jail) creates a new root file hierarchy with a minimal layout (i.e. `/dev`, `/tmp` and `/proc`). This will be the only file hierarchy visible and accessible to any other processes (i.e. subjects) in the jail. The monitor is the only process to keep access to the parent filesystem, in order to extend the jail file hierarchy with new resources.

2. The monitor then bind mounts the initial executable file and needed libraries (cf. Section 4.4) in the jail. The execution of this file then proceeds normally.

3. Throughout the life of the jail, each relevant access attempt to a resource gives rise to a request to the monitor. The monitor then enforces the policy and can extend the jail filesystem when a legitimate access request is received. If needed, the monitor then bind mounts in a read-only or read-write mode a new set of files or directories from the parent filesystem. The monitor completes the filesystem of the jail to make it match the most general reachable domain. Any processes attempting to access a file or directory without requesting it to the monitor will not find the resource if it was not previously mounted.

Access restrictions imposed on subjects rely on two cornerstones. On the one hand, monitor requests must be the only way to gain new accesses. This is guaranteed by the Linux kernel (including the user namespace implementation), which is part of the TCB (cf. Section 2.3). On the other hand, subjects should not be able to compromise a monitor, which is then the crux of a jail security.

It is thus desirable for the StemJail monitor to be developed in a safe system language to avoid common security pitfalls. We choose Rust, a system programming language designed for performance but featuring concurrency and memory safety features [2]. This helps preventing a wide range of recurring security vulnerabilities (e.g. buffer overflow, use after free, dangling pointer, uninitialized memory...). Rust offers low level features like direct memory manipulation or foreign function interface abilities, which are needed to manage low level interfaces such as Linux namespaces, file descriptor manipulation or terminal handling. Moreover, the Rust compiler modularity can include plugins to automatically extend the code, e.g. with serialization. The network protocols used to exchange data between the user client, the portal and the monitors are automatically generated from the program data structure, ensuring a safe parsing.

4.4 Transparent Integration with Applications

When a user logs in to start a session, his shell, be it a text shell or a graphical shell, is configured to automatically launch the portal. To integrate smoothly with the user shell, the easiest way is to replace the existing application shortcuts with wrappers. The StemJail launcher is a wrapper that takes an application with its arguments and requests a jail creation to the portal to launch this application. This way, every application launched by the user actually starts its primary process in a new jail. Using StemJail this way removes the burden of thinking about roles or domains. However, the user can also choose to launch StemJail manually for dedicated sensitive tasks, if needed.

To integrate into an existing system without (static) application modification, StemJail takes advantage of the hook feature from the dynamic linker. StemJail uses the preload functionality to load part of its code into most applications. This preload feature is meant to override the *libc* functions. It applies to dynamically linked ELF binary using the *libc* wrappers around syscalls. Relying on dynamically linked executables is acceptable because classical GNU/Linux distributions only ship dynamically linked ELF binaries.

Obviously, StemJail security guarantees do not follow from wrapping syscalls. Indeed, StemJail is based on a *deny-by-default* principle: only files which have been explicitly exposed by the jail monitor are visible to subjects. However, even though this is not what prevents malicious processes to access files illegitimately, hooking syscalls provides a convenient solution to have complying subjects requesting new accesses to their monitor.

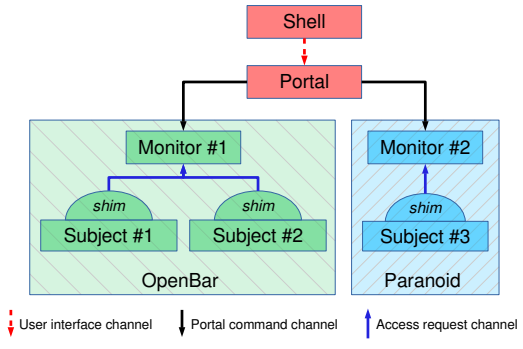

Figure 4: Details of a StemJail Instantiation

The preloaded code, called *shim* (cf. Figure 4), is a shared library including the client part of StemJail and a cache to limit the number of requests. The purpose of the preloaded code is to hook all filesystem related functions (e.g. *open*, *stat*, *rmdir*...) to transform generic applications into StemJail subjects. When a subject performs a syscall via shim, it first goes through its local cache to determine whether a request to the monitor is needed. When deemed required, shim establishes a connection with the monitor through a UNIX socket dedicated to the current jail (in the private `/tmp`) and sends the access request. The monitor can then either evolve to a new domain and bind mount appropriate files, or, when no reachable domain allows the requested access, remain in its current state. The hook then returns and lets the original application syscall go through the kernel. The syscall can thus succeed, if the file or directory is present with appropriate rights, or fail with a "no such file or directory" error from the kernel otherwise. Similarly, raw syscalls placed by malicious processes can only be successful when involving visible resources.

We emphasize that the monitor does not take into account client caches, and it analyzes every request to find out whether it is legitimate. Caches are used to limit the number of requests placed, in order to improve performances. Since the monitor does not mount a resource based on its presence in subject caches, using these latter cannot impact the security of StemJail.

Existing software routinely uses directory listing, which StemJail then needs to enable for transparent integration. This means letting processes retrieve the list of potentially accessible files, and only those. As a result, this is likely to leak metadata on files which might not be reachable later on, but we estimate that it still meets the user security policy. Here is an example of such a corner case. Consider a jail in which the monitor is in current domain `op` (cf. Figure 1), and a subject listing `~/`. Directory `~/Clients/` is visible and should appear. Files beneath `~/Clients/OpenBar/` and `~/Clients/Paranoid/` are accessible through distinct domain transitions, so that the directory `~/Clients/` should list both. However, `~/Company/` should not be listed since it is not currently visible and cannot become accessible through any transition.

The dynamic linker preload feature can be used with an environment variable named *LD_PRELOAD*, with the `ld.so.preload` file or even bypassed with `ld.so.cache` from the `/etc` directory. The environment variable does not require any privileges to be set and is inherited from one execution to another in most cases. The `ld.so.preload` file is located in `/etc`, which is usually owned by the administrator. However, StemJail creates each part of the filesystem of a jail and can hence create this file as well. Moreover, using this file is more convenient than using `ld.so.cache`, because system updates can overwrite `ld.so.cache`. In practice, there are more than 40 function calls that must be hooked, each with their own argument specificities. Each function has been suitably mapped a client request template.

Using the preload feature also accounts for a negligible performance impact. The client code is part of each process; every client can compute the minimum amount of requests to send to the monitor while maintaining a cache of the previous results. The monitor only gets a small number of relevant requests. Moreover, there is no context switching penalty as opposed to other hook methods like *ptrace*.

4.5 User Interaction

StemJail binaries are either installed for the whole system by the administrator, or by a single user for his own benefit. User namespaces need to be activated in the kernel; these are available since Linux v3.8 (released in 2013). StemJail is, as Linux namespaces, architecture agnostic. The configuration is separate from the installation, so any user can use StemJail as he wishes.

As a user can create a role for each of his activities, StemJail must then handle the new, existing and deleted roles. The configuration, including the role and user domain definitions, is taken into account for a domain when it is created and until the end of its life. Any configuration modification is ignored by instantiated domains to avoid inconsistent monitor states.

StemJail is built for the end user and should then use a simple configuration. Each domain is described in a dedicated file in the TOML format. This is a key-value format with sections to group access authorizations. Each section takes a `path` key and a file or directory path as value. By default, every access is read-only. The user makes it read-write by setting the `write` key to `true`. The domain name is extracted from the configuration filename.

StemJail handles two kinds of user interfaces:

- command-line interfaces. A Linux terminal device (TTY) is made up of a master and a slave end. The master end is the user terminal which can send commands (e.g. key pressed, window resizing, interrupt...) and receive text output. The slave end is used by a process (e.g. shell) to communicate with the user. The StemJail portal is used to forward user interactions to subjects through their jail monitor (cf. Figure 4). The portal creates a new pseudo-terminal instance and exposes the slave end into the jail whereas the master end is forwarded to its client (i.e. user terminal).

- graphical interfaces. Graphical applications should not have direct access to the display server to protect from exploiting techniques such as screen grabbing, keyboard sniffing or clipboard spying [12, 22]. The X Window System protocol security is focused on client authentication (across the network) but was not designed to handle multiple untrusted clients. A X proxy such as Xbox [1] or a seamless virtual desktop such as Xpra can be used to create a graphic command firewall between a jail and the user graphic server.

5. FORMAL MODEL OF THE GUARANTEES PROVIDED BY STEMJAIL

5.1 A Partial Order on Domains

We introduce in our formalization a partial order to compare domains. We want to capture the idea that the policy underlying a given domain can be more restrictive than that of another domain. We also use the partial order definition as a pretext to fix notations for concepts introduced in the previous sections. A domain is usually denoted d, while the set of domains is denoted D. For user domains, we rather use u for elements and D_U for the associated set. As D_U is finite, we can write $D_U = \{u_1, \ldots, u_f\}$.

Let us start by ordering objects. Given two paths o and o' in set O, we write $o \sqsubseteq o'$ iff o' is a prefix of o. Intuitively, it means that o refers to a file or directory included in the directory pointed by o'. We recall that accesses are pairs consisting of an action $a \in A$ and an object $o \in O$. The partial order on objects can be applied to accesses as follows.

$$(a, o) \sqsubseteq (a', o') \text{ iff } a = a' \wedge o \sqsubseteq o'$$

It follows the intuition that access (a, o) is more restrictive than access (a', o'), since elements in o belong to o'. We underline that read and write actions are *not* comparable.

In practice, as described in Section 3.2, a monitor decides to grant access requests to subjects if they respect the policy specified by a domain. More formally, an access (a, o) is said to *conform to a domain* iff there exists a rule of the domain exhibiting a more general access:

$$(a, o) \sqsubseteq d' : \exists (r', a', o') \in d' \mid (a, o) \sqsubseteq (a', o')$$

A monitor decision is thus formalized through the following function, that associates a boolean to a domain and an access.

$$\textsc{Access} : (d, a, o) \longmapsto \begin{cases} \text{true iff } (a, o) \sqsubseteq d \\ \text{false otherwise} \end{cases}$$

An access list is said to conform to a domain d if all accesses in the list conform to d.

Rules are triples consisting of a role $r \in R$, an action $a \in A$, and an object $o \in O$. Comparing rules means comparing their accesses, regardless of the role components: a rule refines another if the access referred by the first is lesser than that referred by the second. We can finally provide the definition of the partial order to compare domains as set of rules.

$$d \sqsubseteq d' : \forall (r, a, o) \in d, (a, o) \sqsubseteq d'$$

When respecting rules specified in the lesser domain, a subject enjoys less access rights than another subject following rules contained in the greater domain.

5.2 Automaton for the Evolution of Domains

To perform domain-conforming access control along with dynamic discovery of activity, the monitor is implemented as a stateful process. The monitor state stores the current domain to conform to, which makes it natural to formalize the monitor as an automaton.

As outlined in Section 3.2, the set of possible domains used as states of the monitor matches the set of intermediate domains. Informally, these intermediate domains consist of accesses conforming to several user domains. Therefore, we need to formally define the intersection of two domains. Such an intersection forms a new domain, corresponding to a role which we call r or r'. A rule appears in the new domain if the access it describes is listed in a rule of one of the intersected domains and lesser than accesses listed in rules of the other domain. Thus, the intersection represents the maximal set of accesses conforming to both domains. Formally, the intersection of domains d and d', denoted $d \sqcap d'$, consists of rules (r or r', a_0, o_0) such that:

- either $(r, a_0, o_0) \in d$ and $(a_0, o_0) \sqsubseteq d'$,
- or $(r', a_0, o_0) \in d'$ and $(a_0, o_0) \sqsubseteq d$.

According to this definition, any rule of an intersection of domains is refined by a rule in each intersected domain. As a result, when a monitor authorizes accesses conforming to an intersection of domains, we can rightfully deduce that accesses performed by a subject conform to all domains in the intersection.

As f denotes the number of user domains, we can now define the set $E \subseteq D$ of states of our automaton as the set of all possible intersections of user domains:

$$E = \left\{ \bigsqcap_{i \in J, u_i \in D_U} u_i, J \in \wp([0, f]) \right\}$$

The initial state d_0 of the automaton is the empty domain. We note that $d_0 \in E$, as the intersection of zero user domains.

Let us move on to the transition function between domains implemented in the monitor. When a subject requires access to an object in practice, the monitor must decide whether it grants the request of the subject. As explained in Section 3.2, this decision depends on whether the access conforms to the domain currently stored in the monitor state. If it does, the subject is allowed to proceed. Otherwise, the monitor checks whether there exists a less restrictive intermediate domain to which the access would conform. The set of such domains is described using the following function.

$$\textsc{Next} : (d, (a, o)) \longmapsto \left\{ d' \in E \mid d \sqsubseteq d', \textsc{Access}(d', a, o) \right\}$$

If the access is not in any intermediate domain, the set $\textsc{Next}(d, a, o)$ is empty but the monitor remains in its current state, which we choose to formalize by a possible loop transition. The monitor then refuses the access request from the subject. If the access can legitimately be performed, the monitor transits to the less restrictive intermediate domain amongst $\textsc{Next}(d, a, o)$, and that is its minimal element. We notice that if (a, o) conforms to the current domain d, then this minimal element *is* d. This yields the following definition for the transition function.

$$\textsc{Transit} : (d, a, o) \longmapsto \textsc{Max}(\{d, \textsc{Min}(\textsc{Next}(d, a, o))\})$$

We also use the common infix notation $d \xrightarrow{(a,o)} d'$ for this transition function.

5.3 Proven Security Guarantees

When the monitor transition from one domain to another, all accesses conforming to the source domain remain conforming to the target domain. This property is formalized nicely using the partial order on domains by remarking that traces form ascending chains.

Lemma 1 (Augmenting Transition). *If there exists a trace* $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} ... \xrightarrow{(a_i,o_i)} d_i$ *then* $d_0 \sqsubseteq d_1 \sqsubseteq ... \sqsubseteq d_i$.

Proof. *Let* $d'' \in \textsc{Next}(d, a, o)$*, then by definition we have* $d \sqsubseteq d''$*. Hence,* $d \sqsubseteq \textsc{Min}(\textsc{Next}(d, a, o))$*. As a result,* $d \sqsubseteq \textsc{Max}(\{d, \textsc{Min}(\textsc{Next}(d, a, o))\}) = \textsc{Transit}(d, a, o)$*. Then,* $d \xrightarrow{(a,o)} d'$ *implies* $d \sqsubseteq d'$*, and the conclusion follows.* □

Any access belonging to an intermediate domain, formed by the intersection of several domains, conforms to each intersected domains. The intersection is then lesser than any of its composing domains.

Lemma 2 (Domain Intersection Ordering). *A domain is always lesser than its intersection with another domain:* $\forall d, d' \in D, (d \sqcap d') \sqsubseteq d$

Proof. *Let* $(a, o) \in d \sqcap d'$*, by definition of the intersection of domains, we get that either* $(a, o) \sqsubseteq d$ *and* $(a, o) \in d'$ *or* $(a, o) \sqsubseteq d'$ *and* $(a, o) \in d$*. In both cases,* $(a, o) \sqsubseteq d$*.* □

We can now state the theorem capturing that the monitor enforces the intended security policy. Intuitively, as the monitor is implemented so as to only transition between intersections of user domains, it yields that accesses it may allow to subjects necessarily belong to some user domain. However, this is not sufficient in itself: we want subjects to perform accesses conforming to at least one given user domain all along its execution. Thus, we check that there exists at least *one* user domain where all accesses granted to a subject obey the policy. Therefore, the monitor soundly guarantees that the execution of a subject respects access control restrictions corresponding to at least one user domain, even though the monitor cannot determine which user domain it is from the start of the execution.

Theorem 1 (User Domain Conformance). *Given a trace* $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} ... \xrightarrow{(a_n,o_n)} d_n$*, there exists a user domain* $\Delta \in D_U$ *such that* Δ *is greater than all the states of the trace:*

$$\exists \Delta \in D_U, \forall i \in [0, n], d_i \sqsubseteq \Delta$$

Proof. *Consider a trace of length n,* $d_0 \xrightarrow{(a_1,o_1)} d_1 \xrightarrow{(a_2,o_2)} ... \xrightarrow{(a_n,o_n)} d_n$*. Lemma 1 yields* $d_0 \sqsubseteq d_1 \sqsubseteq ... \sqsubseteq d_n$*. Two cases can arise.*

- *We can first have* $d_n = \emptyset$*. It follows that* $d_0 = \cdots = d_n = \emptyset$*. In this case, we can choose any user domain* $\Delta \in D_U$*, since* $\forall d \in D, \emptyset \sqsubseteq d$*.*
- *Otherwise, we have* $d_n \neq \emptyset$*. Since* $d_n \in E$*, we know that* $d_n = \bigsqcap_{i \in J} u_i$ *for some set* $J \in \wp([0, f])$*. Moreover, since we know* $d_n \neq \emptyset$*, then* $J \neq \emptyset$*. We let* $\Delta = u_{i_0}$ *for* $i_0 \in J$*. We know* $u_{i_0} \in D_U$ *so* $\Delta \in D_U$*. Since* $d_n = u_{i_0} \sqcap (\bigsqcap_{i \in J, i \neq i_0} u_i)$*, Lemma 2 yields* $d_n \sqsubseteq u_{i_0} = \Delta$*. The conclusion follows for the whole chain:* $d_0 \sqsubseteq d_1 \sqsubseteq ... \sqsubseteq d_n \sqsubseteq \Delta$*.*

□

6. VALIDATION

6.1 Use Case Scenario

We go back to our running example to illustrate how different activities are impacted by the presence of StemJail. We sketch out a simple and classical user workflow scenario: the consultant starts a file explorer and browses to the file of a specific client he wants to work for. He then opens the client file with the PDF reader `Viewer`. Later on, `Viewer` gets compromised, and a malicious process then tries to exfiltrate data from his system. Traces 1 through 5 show extracts of logs generated by StemJail components during this scenario.

Trace 1 shows the user launching the file explorer through the StemJail launcher, as described in Section 4.4. The launcher connects to the portal daemon and requests a jail creation. Consequently, it spawns a monitor instance with the most general current domain as a current state, namely `opc`, which stands for the disjunction `OpenBar` *or* `Paranoid` *or* `Company`. This monitor then creates a working directory to prepare the filesystem initially exposed in the jail. This

```
Loaded configuration: profiles: ["OpenBar", "Paranoid", "Company"]
Portal got request: Run(DoRun(RunRequest { profile: None, command: ["/usr/
    ↪ bin/file-explorer"] }))
Running jail: OpenBar || Paranoid || Company
Child jailing
Creating tmpfs in /proc/fs/nfsd
  Bind mounting from /usr to /proc/fs/nfsd/usr
  [...]
Creating tmpfs in /tmp
Populating /dev
  Creating tmpfs in /dev
  Bind mounting from /dev/null to /proc/fs/nfsd/dev/null
  [...]
  Creating tmpfs in /dev/shm
Pivot root
Got jail PID: 2
Waiting for child 2518 to terminate
```

Trace 1: Jail Initialization

hierarchy is built on a temporary filesystem (`tmpfs`) to be able to add files or directories when needed. The monitor then populates the jail filesystem with the necessary files (`/tmp`, `/dev`...) as explained in Section 4.3. Finally, the file explorer runs as a new process in the newly created jail. The user is now able to access documents from all domains through a domain transition.

```
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home", write: false } }))
  No domain reachable: access denied
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user", write: false } }))
  No domain reachable: access denied
```

Trace 2: Denied Access Requests Before Subject Compromise

Trace 2 shows access requests which the monitor does not honor by the requested mount operation. Here, the monitor does not mount directories `/home` or `/home/user` because there is no domain amongst those defined in which these accesses are allowed.

```
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user/Clients/OpenBar", write: false } }))
  Bind mounting from ./parent/home/user/Clients/OpenBar to ./
      ↪ tmp_mount_zLpqpb2ojEGn/
  Moving bind mount from ./tmp_mount_zLpqpb2ojEGn to /home/user/Clients/
      ↪ OpenBar
  Removed ./tmp_mount_zLpqpb2ojEGn
  Domain transition: OpenBar || Paranoid || Company -> OpenBar
  Access granted to [AccessData { path: "/home/user/Clients/OpenBar", write
      ↪ : true }]
```

Trace 3: Domain Transition

Trace 3 shows the user browsing to open a document in `~/Clients/OpenBar/`. The `Viewer` application sends an access request for this path to the monitor. This time, there exists a domain, `OpenBar`, reachable from the monitor current domain, in which this access is listed. Moreover, since the access is only available in this particular domain, it is the most general possible transition. The monitor therefore triggers the transition to the `OpenBar` domain and bind mounts all the new available paths (which at least include the requested path) from the parent filesystem to the jail filesystem. As mentioned in Section 4.2, to avoid race-conditions, the mount operations are first performed in a temporary directory and then moved to the jail.

```
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user/Clients/OpenBar/.", write: false } }))
  Current domain already allows this access
  Access granted to []
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user/Clients/OpenBar/malicious.pdf", write: false } }))
  Current domain already allows this access
  Access granted to []
```

Trace 4: Authorized Access Requests

Trace 4 shows legitimate access requests which are already granted by the `OpenBar` domain. The subjects caches (cf. Section 4.4) are stored by thread, so access requests can be sent to the monitor before these caches are filled. We see that a subject (`Viewer`) successfully requests for access to a PDF file.

```
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user/Clients/.", write: false } }))
  No domain reachable: access denied
Monitor got request: Shim(Access(AccessRequest { data: AccessData { path:
    ↪ "/home/user/Clients/Paranoid", write: false } }))
  No domain reachable: access denied
```

Trace 5: Denied Access Requests After Process Hijacking

We emphasize that StemJail does not detect malicious processes. It is rather treating all processes as potentially malicious. Trace 5 illustrates what happens after the `Viewer` get compromised by a malicious file within the `OpenBar` domain. The compromised process can then access the `OpenBar` data (to leak it or encrypt it if it is a ransomware). The logs show the attempts of the malware at having the monitor mount resources of `Paranoid`, on behalf of the StemJail client which was hijacked. Since there is no possible transition to a domain which would enable these bind mounts, the monitor declines the requests. We recall that the malware would not be more successful in performing raw syscalls: the files are not exposed within the jail in which it runs.

The monitor stops when all its subjects end or when the user explicitly sends a request to the portal. The user can thus kill all the processes from a jail at will.

6.2 Review of Common Security Issues

As explained by Garfinkel [13], some typical security issues are commonly found in sandbox implementations. This list summarizes interesting pitfalls and explains how StemJail addresses them.

Common defects in sandboxes include *the incorrect mirroring of the OS state*, *incorrect replication of OS code* and *unexpected side effects of denying system calls*. Contrary to what is classically done in sandboxes, StemJail security does not rely on filtering out syscalls or tampering with their arguments based on decisions taken while maintaining a local copy of the OS state. Indeed, our implementation does not forbid syscalls to go through with their original arguments. StemJail rather has the side effect of extending the filesystem view of subjects based on their requests. As the current jail filesystem representation and reachable domains constitute the only information relevant to our security policy enforcement, it yields a fairly simple state to maintain in the monitor. In addition to that, the monitor is the only process modifying this filesystem view, and does so atomically. Last but not least, the monitor is the only piece of code responsible for enforcing its policy, in the following sense. Firstly, all visible resources in a jail have been mounted by the monitor via syscalls it placed itself. Secondly, the monitor does not emulate the kernel. The only way in which it relies on the kernel behavior is in that a non-visible resource should not be accessible.

Another kind of flaws stems from *overlooking indirect paths to resources*. This is used to create unwarranted communication channels between subjects supposedly restricted by security policies. StemJail somehow indirectly addresses this kind of issues, by guaranteeing that the filesystem created by the monitor is the only shared resource between jails. As

a result, shared resources that subjects can acquire depend on the user policy expressed by the domains. For example, the policy should not share UNIX sockets between domains that should not be able to cooperate.

Other common oversights such as time-of-check-time-of-use-based vulnerabilities (TOCTOU [7]) are due to *race conditions*. StemJail prevents this kind of flaws by using a unique monitor thread to manage and ensure consistent domain transitions sequentially and reliably, thanks to Rust built-in concurrency properties. In this thread, as explained in Section 4, a special attention has been taken to perform atomic operations. Other races like *argument races* are irrelevant to StemJail because there is neither syscall interposition nor manipulation for access requests, which are entirely copied to the monitor. A syscall will naturally return an error if no previous request has made the requested resource visible.

6.3 Performance Impact

To evaluate the performance of StemJail, we present in Table 1 benchmarks comparable to those used by Potter et al. [19] and Kim et al. [17]. The methodology used is described in a script provided with the StemJail source code[1]. The system used for the experiments is an Intel Xeon 2GHz using 2 cores with hyper-threading enabled (i.e. 4 logical cores), 4GB RAM (DDR2, 667 MHz), running Debian Linux with a 4.4 kernel. For each benchmark, we provide performance figures related to execution on a mechanical hard drive (HDD column – to enable comparison with other benchmarks) and execution in a *tmpfs* mount point (RAM column – to remove the impact on the results of mechanical hard drive and cache).

The results are consistent with the expected behavior of StemJail. The most important overhead of StemJail is related to the use of the subject caches.

The Gunzip benchmarks (line 1 of Table 1) stress the filesystem for read operations from the archive and write operations to the original file. Except for the first accesses (including the archive file), there is no communication with the monitor, hence no visible penalty.

The Untar benchmark (line 2 of Table 1) exhibits a low performance penalty except when launched on a *tmpfs* filesystem. Indeed, extracting a lot of files to the filesystem requires intensive use of the subject's cache. However, the overhead due to StemJail remains low in the HDD case, where the mass storage device is the real performance bottleneck.

The Zip benchmark (line 3 of Table 1) involves a lot of file operations like Untar, but implies browsing and reading the content of all files. The intense computation related to the decompressing (which is not present with Untar) is the bottleneck and masks the filesystem read operations, even for a *tmpfs* filesystem.

Building a kernel with consecutive processes (line 4 of Table 1) creates a lot of short-lived processes accessing a lot of different files. We use the default configuration for the `x86_64` architecture. The results show a minimal impact while compiling in a jail. The fifth benchmark (line 5 of Table 1) is obtained by building a kernel with one parallel job per logical CPU core. Thanks to an efficient cache in each process, the StemJail monitor does not get a lot of requests and the performance penalty remains constant.

[1] https://github.com/stemjail/stemjail/blob/master/tools/bench.sh

According to the performance benchmark from recent *ptrace*-based sandboxing [17] (optimized with Seccomp-BPF), which seems to have been run on a hard drive, the StemJail mechanism is three to five times more efficient. This practical result confirms the benefit of loading some code on the subject side to avoid unnecessary transactions.

7. PERSPECTIVES

7.1 Related Work

SELinux and the other current Linux Security Modules (LSM) are designed to be managed by administrators, which does not match our requirements. The same goes for Linux compartmentalization solutions, such as LXC, Docker or VServer. Even if they employ user namespaces, *root* helpers are still required for a full container because user namespaces do not provide all the features they need.

Qubes OS has a good security architecture and can reduce the attack surface of the machine by using stubdomains [22]. However, its main drawbacks are that it cannot be integrated in a common GNU/Linux distribution and that it is designed for a single end user, which is also an administrator.

Unlike Ostia [14], StemJail uses a shared library (shim) to send requests to an external trusted process but does not emulate syscalls for access control, only for some directory listing (cf. Section 4.4). Moreover, the StemJail shim uses a cache to only send requests when strictly needed, which significantly lowers the overhead. Dynamic user-oriented sandbox such as Mbox [17] helps to isolate some dangerous activities but are designed for power users. The other sandbox mechanisms such as Oz or Subuser are promising, but request significant user interaction.

While a few interesting user-oriented projects do exist, up to our knowledge, none of them provides an automatic dynamic compartmentalization feature like that of StemJail. Moreover, the programming language properties brought by Rust are essential for secure compartmentalization software.

7.2 Limits of the Approach and Future Work

Activities can overlap. If one domain is strictly included in another, the monitor will never enforce the most restrictive policy. For example, we suppose that an activity requires access to `/a/b`, whereas another requires access to `/a`. Since accessing `/a/b` is possible in both domains, such a request will not make the monitor transition to the first domain once in the second one. The solution to this problem is to provide more input than paths to the monitor (e.g. coming from the user).

StemJail deals with paths to match files in the policy. These objects could be generalized further to add other kind of resources like network objects to the security policy. To define these access rules, an interesting generic approach could be to define objects with URI in place of paths. This could help expressing a wide range of possible access control.

A web browser is nowadays a shell able to access a wide range of diverse information through websites. Some browsers (e.g. Chromium [4]) benefit from a compartmentalized architecture, which renders them compatible with the use of StemJail components. Though some integration efforts still remain before these web browsers feature StemJail, we believe that it constitutes a particularly relevant use case for our approach.

Task	Normal HDD	Normal RAM	StemJail HDD		StemJail RAM		Description
Gunzip	16.04s	6.13s	16.03s	0.0%	6.13s	0.0%	Decompressing the Linux 4.4 archive
Untar	68.30s	1.57s	69.95s	2.4%	1.97s	25.4%	Extracting the Linux 4.4 archive
Zip	36.36s	31.76s	38.42s	5.6%	33.49s	5.4%	Compressing all files of Linux 4.4
Build (-j1)	1137.38s	1134.10s	1190.16s	4.6%	1188.02s	4.7%	Compiling Linux 4.4 with 1 job
Build (-j4)	320.96s	315.09s	344.69s	7.3%	330.37	4.8%	Compiling Linux 4.4 with 4 parallel jobs

Table 1: Results of Performance Benchmarks (StemJail 0.4.0)

8. CONCLUSION

This paper presents StemJail, a userland isolation mechanism that automatically enforces separation of information according to user-specified needs.

Our first objective is to seamlessly integrate relevant access control in the user workflow, to improve usability over traditional static policies. This objective is met by automated role discovery, which progressively deduces a user activity and creates custom jails accordingly. Moreover, our solution aims to allow any user of the system to create and use his security policies to mitigate the consequences of the execution of malwares. Thus, policies must be simple enough to be set up and understood by the user. This second objective is achieved thanks to our proposal based on the organization of his storage space in directories. We have formalized our approach to carry out a proof of the security guarantees enforced throughout the life of the jailed processes.

We avoid an increase of the attack surface by taking advantage of the Linux user namespaces, instead of adding more code to the kernel. Only introducing a low performance overhead, the open source project StemJail can be efficiently used to secure machines running on Linux, since compatibility with current applications is ensured.

9. REFERENCES

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *USENIX Security Symposium*, 2000.

[2] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the Servo Web Browser Engine using Rust. In *ICSE*, 2016.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A domain and type enforcement UNIX prototype. In *USENIX Security Symposium*, 1995.

[4] A. Barth, C. Jackson, C. Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser, 2008.

[5] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE Corp., 1973.

[6] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual servers and checkpoint/restart in mainstream Linux. *ACM SIGOPS Operating Systems Review*, 2008.

[7] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing systems*, 1996.

[8] D. Blazakis. The Apple Sandbox. In *Black Hat DC*, 2011.

[9] D. F. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Security and Privacy*, 1989.

[10] W. Drewry. Dynamic seccomp policies (using BPF filters), 2012. https://lwn.net/Articles/475019/.

[11] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *NIST-NCSC*, 1992.

[12] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC*, 2005.

[13] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, 2003.

[14] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *NDSS*, 2004.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications (Confining the wily hacker). In *USENIX Security Symposium*, 1996.

[16] N. Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

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

[18] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *ACSAC*, 2003.

[19] S. Potter and J. Nieh. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *USENIX Annual Technical Conference*, 2010.

[20] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, 2003.

[21] M. Russinovich. Inside windows vista user account control. *Microsoft TechNet Magazine*, 2007.

[22] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. 2010.

[23] M. Salaün. StemJail source code, 2015. https://github.com/stemjail.

[24] Z. C. Schreuders, T. McGill, and C. Payne. The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls. *Computers & Security*, 2013.

[25] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp. Confining root programs with domain and type enforcement. In *USENIX Security Symposium*, 1996.

[26] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium*, 2010.