University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jędrzej Jabłoński and Marcin Pawłowski
Student no. 248386, 248272

Secure sandboxing solution for GNU/Linux

Master's thesis
in COMPUTER SCIENCE

Supervised by
Janina Mincer-Daszkiewicz, Ph. D.
Institute of Computer Science,
University of Warsaw, Poland

December 2011

Supervisor’s statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfills the requirements for the degree of Master in Computer Science.

Date Supervisor’s signature

Authors’ statement

Hereby we declare that the present thesis was prepared by us and none of its contents was obtained by means that are against the law. The thesis has never before been a subject of any procedure of obtaining academic degree. Moreover, we declare that the present version of the thesis is identical with the attached electronic version.

Date Authors’ signatures

Abstract

In this thesis a new sandboxing solution for GNU/Linux systems, designed and implemented by authors, is presented. Existing solutions are described, and the gap in the market is pointed out in the requirement analysis. Created solution is based on Linux kernel mechanisms (i.e. cgroups, rlimits, namespaces, capabilities, secure bits), which were thoroughly explained. Porting existing software to this sandboxing tool is described on the example of Apache suEXEC module. Finally, main difficulties of implementation are analyzed, and the results of performance tests are demonstrated.

Keywords

arbitrary code execution, capabilities, cgroups, chroot, container, DAC, kernel, Linux, MAC, PID namespace, rlimit, sandboxing, secure execution, security, suexec, VFS namespace

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

D Software
D.4 Operating Systems
D.4.6 Security and protection
Security kernels

Title in Polish

Bezpieczne rozwiązanie typu piaskownica w systemie GNU/Linux

Contents

1. Introduction **7**
1.1. Basic security problems 7
1.2. Gap in the market 7
1.3. Project goals 7
1.4. Final product 8
1.5. Structure of this paper 8

2. Definitions **9**

3. Existing solutions **11**
3.1. Overview 11
3.2. System wide solutions 11
3.2.1. Discretionary access control 11
3.2.2. Mandatory access control 12
3.2.3. Role based access control 15
3.2.4. Multilevel security 15
3.3. Per application solutions 16
3.3.1. Setuid/setgid 16
3.3.2. Chroot 17
3.3.3. Capabilities 17
3.4. Summary 18

4. Requirements analysis **19**
4.1. Functional requirements 19
4.2. Design requirements 20
4.2.1. Kernel level 20
4.2.2. C Library 22
4.2.3. SUID supervisor 23
4.3. Adapting existing software 24
4.3.1. Overview 24
4.3.2. Specific requirements 25
4.4. Summary 25

5. System architecture **27**
5.1. Overview 27
5.2. Available kernel mechanisms 27
5.2.1. Rlimits 27
5.2.1.1. Overview 27
5.2.1.2. Resources 28

5.2.2. Cgroups . . . 31
5.2.3. Namespaces . . . 32
5.2.4. Capabilities . . . 32
5.2.4.1. List of capabilities . . . 32
5.2.4.2. Examples of capability equivalence . . . 34
5.2.4.3. Capability bounding . . . 35
5.2.4.4. Security bits . . . 35
5.3. Mechanisms used in sandboxing library . . . 35
5.4. Summary . . . 36

6. Implementation **37**
6.1. Overview . . . 37
6.2. Public API . . . 37
6.3. C library implementation . . . 40
6.4. VFS resources used by sandbox . . . 48
6.5. Supervisor . . . 49
6.6. SUID Wrapper . . . 51
6.7. Porting suEXEC . . . 52
6.7.1. suEXEC architecture . . . 53
6.7.2. Sandboxing in suEXEC . . . 54
6.7.3. Patch . . . 54
6.8. Summary . . . 56

7. Tests **57**
7.1. Testing platform . . . 57
7.2. Correctness tests . . . 57
7.2.1. Memory limits . . . 57
7.2.1.1. Single process test . . . 57
7.2.1.2. Multiple processes test . . . 58
7.2.1.3. Results . . . 59
7.2.2. CPU usage limits . . . 59
7.2.2.1. Tests description . . . 59
7.2.2.2. Results . . . 59
7.2.3. Device access . . . 61
7.2.3.1. Results . . . 61
7.2.4. VFS restrictions . . . 62
7.2.4.1. Results . . . 62
7.3. Performance tests . . . 62
7.3.1. Cost of execution . . . 62
7.3.2. Cost of cgroup memory limiting mechanism . . . 63
7.3.3. Cost of supervisor and cgroup CPU accounting . . . 64
7.3.4. Cost of VFS namespaces . . . 64
7.4. suEXEC tests . . . 65
7.5. Summary . . . 65

8. Summary **67**
8.1. Goals achieved 67
8.1.1. C library 67
8.1.2. SUID wrapper 67
8.1.3. Supervisor 68
8.1.4. suEXEC patch 68
8.1.5. Performance 68
8.2. Further development 68
8.3. Authors' contributions 69

A. Attached CD **71**
A.1. Sandboxing solution source (*src/*) 71
A.2. Apache suEXEC module patch (*external/*) 72
A.3. Documentation (*doc/*) 72
A.4. Tools used for testing (*tests/*) 72

Chapter 1

Introduction

1.1. Basic security problems

Security of computer systems was a serious concern from the very beginning of their development. Errors have always been inevitable, but wrong software architecture catalyzes their escalation. High-end solutions solve this problem mostly by reducing bug rate, which is effective only with huge budget spent on it. As user applications became more complex and the Internet became more popular the general approach to the security problem had to change.

The rule of thumb in modern security models is: never give more resources to the program than absolutely necessary. However, convincing the rule may seem, it is not the user who directly decides about the resources, but the operating system. Although GNU/Linux offers many tools to restrict the usage of CPU, memory, virtual filesystem, cross-process communication etc., they are all designed to be used by software engineers and system administrators rather than end-users.

1.2. Gap in the market

Importance of secure arbitrary code execution increases with development of Internet technologies. Two basic concepts of Web 2.0 (interactive webpage browsing and information sharing) demand a dangerous combination of code execution on the user's side and granting access to user's private files. The first approach to security was to keep the programming languages simple. This, however, provides only short-term security. The interpreters evolved in direction of speed optimization, which in most cases does not harmonize with security. Moreover the development of Web 2.0 and the concept of Web 3.0 place new demands on functionality of these languages.

On the other hand, one can see that the personalization of web technology will soon significantly raise the need for massive arbitrary code execution on the server's side. It is already an issue in stock webservices and the solution is again the same dead end (simplification of the language). The other example is webpage hosting in which case the problem is mostly solved by legal agreements or CPU-consuming solutions such as virtualization.

1.3. Project goals

The goal of this project is to supply GNU/Linux users with a three-layer tool (kernel mechanisms + library + interface) which allows them to restrict access to resources for the processes they are running. Depending on the perspective this can be seen as a process supervision or low-level isolation framework.

From user's point of view what we want to achieve is a grand unification of supervisor tools. Ubiquitous web technology provides numerous mechanisms of the same security structure: supervisor (web browser, Java VM, Flash player) runs a "partly trusted" web application. This inconsistent model causes dissipation of responsibility and as a result makes the user agree to trust each and every supervisor used on the web. Our sandboxing technology aims to let the user take control over what is being accessed by applications and to make the arbitrary code execution harmless. The design itself is supposed to make the security independent of the web technology.

For the programmer the concept is equivalent with kernel-level isolation, though not all functionalities are implemented. It is, however, the goal of this project to provide proper isolation in areas of process namespaces, CPU and memory usage, virtual filesystem. The library is designed to be used by programmers to tune sandboxing setting for some specialized cases (such as suEXEC module in Apache) or to provide integrated interface in larger systems (such as web browsers).

1.4. Final product

The final product consists of a three-layer mechanism (C library and process supervisor) and examples of usage (exploits and suEXEC module patch). The program can be used in two basic ways: either as a SUID-root application which uses the library to set the sandbox and run a given subapplication in it, or indirectly as a library. The first use case is demonstrated by a set of exploits which can be turned harmless when put inside a sandbox. The second is presented in the suEXEC patch, which makes running user's code secure.

1.5. Structure of this paper

Firstly, existing solutions will be presented in details. After the reader is familiarized with these solutions, it will be clear what the gap in the market is. A closer look at the requirements and use-cases will lead to the system architecture. Afterwards, main difficulties in the implementation will be explained. It will be then demonstrated how the solution can be applied in an existing piece of software on an example of Apache suEXEC module. Finally results of correctness and performance tests will be presented.

Chapter 2

Definitions

In further chapters of this paper we will use terminology which might be inscrutable for the reader if the explanation hadn't been provided. The most frequently used expressions whose meaning might be unclear are described below:

access control mechanism which controls possibility of interaction with a resource,

address space an abstract view of the computer memory accessible to a process,

application computer software designed to perform specific tasks,

arbitrary code execution ability of an attacker to execute any commands in any order,

BNF Backus Normal Form notation for context-free grammars,

capability by 'capability' we mean POSIX capability (it should not be confused with capability-based security model),

CGI Common Gateway Interface — method for web servers to delegate generation of web pages to executable files,

denial of service (DOS) situation where a resource is unavailable to its intended users due to an attack,

effective uid/gid this uid/gid is used to evaluate privileges of the process to perform particular actions,

environment set of variables that are passed to the process during its execution,

gid unique identification number of a Unix group,

namespace a container for system resources or subjects which allows to transparently isolate them,

nice kernel call that sets the priority based on niceness,

NFS Network File System — a network file system protocol,

object a resource (or its abstract representation) in a computer system,

OOM killer Out Of Memory killer — kernel mechanism used to terminate processes in order to gain free memory,

PAM Pluggable Authentication Modules — mechanisms to integrate authentication schemes into a high-level API,

POSIX Portable Operating System Interface for Unix — a set of standards specified by the IEEE for compatibility between operating systems,

priority property of a process that is taken into consideration by the scheduler — lower priority values cause more favorable scheduling,

process an instance of a computer program that is being executed,

subject an entity which performs actions on a computer system, usually process or thread,

super user privileged subject who has rights to access and manage all resources of the given system,

target an object or subject that will be accessed as a result of an action,

uid unique identification number of a Unix user,

VFS Virtual File System — an abstraction layer on top of specific filesystems support.

Chapter 3

Existing solutions

3.1. Overview

This chapter will cover some of the existing security solutions. Its goal is to make the reader familiar with the tools already available, their scope, power and flexibility. It will also try to depict their specific shortcomings which made the authors confident that there is a need for yet another solution, tackling the problem from a different angle. The chapter is divided into two parts, each of them providing information about different kind of available mechanisms.

3.2. System wide solutions

Wide variety of tools fall into this category. Their common denominator is the fact that they affect almost every single operation that happens on a given computer. The great strength of this approach is that it gives a holistic solution which can be leveraged to build very sophisticated tools. On the other hand the complexity of interactions between applications and operating system is growing each year. This leads to a situation when adjusting system wide security solution to every single application that is being used, becomes very difficult. Also, as holistic security mechanisms have such wide impact, they often produce hard to predict consequences. That is why,inside they reside mainly in the domain of system administrator and vendors of operating systems. Below we will shortly cover a few such solutions.

3.2.1. Discretionary access control

In this doctrine (also abbreviated as DAC) access to objects is granted or denied based on the identity of subjects (or groups) who own the object. In other words, each access right can be granted or removed by the subject which owns the object. The name comes from the fact that a subject which is capable of accessing a resource can pass this ability to other subjects on its own discretion.

The model is implemented for example by Unix semantics of owner and access permissions for filesystem objects. Each file in Unix OS has a user and a group associated with it - the owner (identified by uid number) and the group owning it (identified by gid number). From now on, we will refer to them as 'owner' and 'group'. The owner can set three types of access restrictions - to read, to write, and to execute. All subjects are divided into three categories: the owner, members of the group owning the file and all the others. Owner can limit types of access allowed for each group. For example, owner can set that he can read/write/execute, the group can read and write and the rest of the subjects can only read the file.

This model is very simple, thanks to which it is easy to understand and debug. What is more, it is fairly powerful, even in its simplest form. It has gained very widespread adoption in server systems. However, the lack of flexibility is obviously limiting: the division of subjects into only three categories makes it very hard to define fine grained rules. The mechanism of access control list (ACL) was invented to extends the basic DAC — it allows the owner to grant access right to arbitrary subjects/groups on a per file basis.

Even with this extension, the model is still not sufficient to provide a complete security solution. One of the biggest deficiencies of this approach is that it suffices for the attacker to subvert a process performing any operation to gain access to all the resources which can be possibly accessed by the legitimate subject owning the process.

3.2.2. Mandatory access control

In this model (abbreviated as MAC) subjects are not entrusted with full control of the resources they own. The security rules enforced by an operating system define which operations are allowed. Subjects may not be allowed to pass their access rights to other subjects. This is usually implemented by attaching attributes to all subject and objects and building a set of rules which specify combinations of attributes of the initiator and target that should lead to allowing the action to be performed. In this case the security of the entire system is managed centrally by a single database which holds rules. Individual subjects (not administrators) are not able to override the security mechanisms and have no role in defining it.

The core strength of such approach is its flexibility. With proper tools one can possibly invent any security policy and enforce it on the entire system. On the other hand, the complexity of managing every single operation that can be performed is tremendous. State-of-art applications have many millions of lines of code and their users are not aware of all the reasonable interactions they were programmed to perform. This leads to misconfiguration of the MAC system and spurious errors. The best exemplification of this situation is SELinux (as described in [13]) .

SELinux is a MAC system developed for Linux kernel by the National Security Agency (its goal was to provide security following the model defined in [12]). It works on top of a traditional Unix DAC system — a request to be accepted must first pass SELinux policy and later it is checked against the usual DAC rules. SELinux attaches a context to each subject. The context consists of an user, a role and a domain. Furthermore, each system object or resource (file, device) also gets a context which consists of a name, a role and a type. The policy usually consist of explicit rules which allow access based on source context (the subject's) and destination context (the object's one). SELinux has two fundamental modes of operation: enforcing and permissive. In enforcing mode all actions that were not specifically allowed in policy are rejected. In permissive mode all actions that were not explicitly disallowed in policy are allowed.

SELinux is a very powerful MAC system which can control almost any action performed by the OS for the subjects. However, the system is very complex, each interaction must be explicitly allowed in enforcing mode, on the other hand permissive mode provides much less security against new, unexpected attacks. In the end it is very difficult to design a complete security policy for a subject executing given application. Thus vendors often do not bother to design policies for their products and simply require the OS to disable SELinux. What is more even OS vendors prepare policies which only affect carefully selected set of crucial applications and let all the rest of the system to simply avoid the MAC mechanism. To achieve that all subjects which are not to be overseen are put in special "unrestricted" domain, which allows all actions.

The power of SELinux can be easily noticed when one analyzes a policy file. Below a short intercept, from a policy for a subject running BIND (a well known DNS server), is presented. It depicts both the complexity of creating proper rules, as well as the flexibility of SELinux. Each line

of code is described by a comment (starting with '#') preceding it. Comments contain numbers in parenthesis, which will be used later on to refer to the code lines following them.

```
# Rules for the named_t domain.

#define types of objects used by the named_t domain
type named_port_t, port_type;
type rndc_port_t, port_type;

#name of the domain
daemon_domain(named)

# (1) objects which can be executed by subject in named_t domain
can_exec(named_t, named_exec_t)

# (2) allow reading directory contents (names of files)
allow named_t sbin_t:dir search;

# (3) set scheduling info
allow named_t self:process setsched;

# named configuration file types
type named_conf_t, file_type, sysadmfile;
type rndc_conf_t, file_type, sysadmfile;

# master zone file types
type named_zone_t, file_type, sysadmfile;

# slave zone file types
type named_cache_t, file_type, sysadmfile;

# (4) allow reading of files in /etc
allow named_t etc_t:{ file lnk_file } { getattr read };
allow named_t etc_runtime_t:{ file lnk_file } { getattr read };
allow named_t resolv_conf_t:file { getattr read };

# (5) Named can use network devices
can_network(named_t)

# (6) allow UDP transfer to/from any program
can_udp_send(domain, named_t)
can_udp_send(named_t, domain)
can_tcp_connect(domain, named_t)

# (7) Bind to the named port.
allow named_t named_port_t:udp_socket name_bind;
allow named_t { named_port_t rndc_port_t }:tcp_socket name_bind;

# (8) read configuration files
```

```
r_dir_file(named_t, named_conf_t)

# (9) read/write dynamic zone files
rw_dir_create_file(named_t, named_zone_t)
allow named_t named_zone_t:file setattr;

# (10) write cache for secondary zones
rw_dir_create_file(named_t, named_cache_t)


# (11) read /proc/cpuinfo.
allow named_t proc_t:dir r_dir_perms;
allow named_t proc_t:file r_file_perms;

# (12) read /dev/random.
allow named_t device_t:dir r_dir_perms;
allow named_t random_device_t:chr_file r_file_perms;
```

As BIND is a DNS server it needs to perform at least the following operations:

- read its generic configuration files and system network configuration — rule (4) and (8),
- read and write file with data about DNS mapping (these can change during runtime, so read access is not sufficient) — rule (9),
- DNS sec reply contains pseudo random numbers. To grant the daemon access to OS-provided source of randomness, it is allowed to access /dev/random — rule (12),
- use the network device — rule (5),
- reply to request over UDP protocol unless the reply is bigger than 512 bytes, then it sends them over TCP (also zone transfers are done over TCP) — rule (6) allows for this,
- listen for requests on port 53 (bind the port in Unix terminology) — rule (7).

Furthermore due to BIND implementation it needs to perform the following actions:

- read the information about the server's CPU — rule (11),
- set its own scheduling priority (as used by the OS scheduler) — rule (3),
- execute specific subprocesses — only files with *named_t* or *named_exec_t* attribute can be executed by BIND — rule (1),
- read and write files used for caching frequently used mappings — rule (10).

This is only a small fragment of the entire policy file for BIND. As one can see, preparing it requires knowing deep knowledge of subject's implementation details. One has to know all correct access requests the subject may possibly make. On the other hand, the power of the mechanism is also clearly seen — the subject will not be able to access any file or device that is not mentioned (by its extended attributes) in the policy — of course this is only true when SELinux is run in enforcing mode.

3.2.3. Role based access control

Role Based Access Control (from now on referred to as RBAC) is a specific form of MAC. As described in [11], RBAC puts main stress on the idea of role. The intuition behind the term 'role' comes from everyday life: person performs activities that are allowed within his role in society. Similarly, in RBAC one creates a role associated with a specific set of closely connected tasks. A solid example of such role may be configuring a system daemon (for example HTTP server) or managing logging information gathered in /var/log directory. Of course this does not differ in terms of expressive power from MAC systems mentioned above. However, thanks to using different ideas as a base, it may be easier to map real world requirements into an RBAC policy. Generally this kind of approach is used in big organizations with rigid hierarchy and well defined responsibilities of people.

One of the most widespread RBAC system available for personal computers is grsecurity for Linux kernel. It is a set of patches that add RBAC features to the kernel, as well as makes some minor changes aimed at improving general security of the kernel. In grsecurity roles are divided into following categories:

- user roles,
- group roles,
- special roles,
- domains.

The first two are direct mappings from the traditional Unix users and groups, i.e. user *xyz* is assigned roles *xyz* and *abc* when *xyz* belongs to group *abc*. Special roles are created in order to allow users to perform administrative actions. *httpd_admin* role may be used to allow specific users to manage the Apache web server. In order to take advantage of the special role a subject (which normally runs only with user/group role) must perform a transition. The policy file describes how and when this can happen. Usually there are at least two types of transitions: uid/gid based only and uid/gid based with verification. In the first one a subject with given uid/gid may simply perform transition with no more obligations. The second one requires the subject to prove identity, for example by providing the password for the given uid. In order to make the administration easier grsecurity provides configuration option which allows PAM to perform identity verification. Domains are used to group roles which have no common gid/uid or group special roles.

In order to make roles administration easier grsecurity provides a mechanism known as role inheritance. It allows some roles to automatically get all the abilities of some other roles. For example administrator role is a specific incarnation of the user role (with much greater number of allowed actions). Again the hierarchies of roles map naturally into the structure of real world organizations.

In order to make policy building a simple and reliable task grsecurity provides tools which record the activities of a process and build a role, based on it. The tools try to minimize the capabilities needed to preserve the original functionality.

As one can easily notice the general guidelines and use cases for both SELinux and grsecurity do not differ much. This is an exemplification of the fact that RBAC is a specific incarnation of generic system-wide MAC.

3.2.4. Multilevel security

This particular approach is not concerned with the system security per se, but deals with the flow of information in the system. The purpose of multilevel security (referred to as MLS) is to prevent the leakage of secret information to subject with no clearance for it. In order to ensure that, MLS defines

levels of security clearance and set of rules which govern the flow of information between them. Generally, subjects may read information which has level as high as theirs or lower. On the other hand they are only allowed to write data into object which will get at least their level of clearance. This means, that high level subject can read public data of a low level subject, but is unable to send any information to him. In order for the high level subject to be able to share information with a subject that is on a lower level, the information must pass through a process called clearance — someone must verify that the information is not sensitive.

3.3. Per application solutions

These solutions affect only a single subject (possibly all subjects that he subsequently creates). Thanks to such approach, tools should be easier to understand and control, as only a small part of the system is affected. Of course as this techniques concentrate on single entities, they lose a lot of flexibility of system wide tools. Furthermore, most of the tools belonging to this category are used to prevent only a single type of threat. It is much more difficult (or even impossible) to control interaction between different entities using only per application security solutions. As such per application tools are usually only a necessary add-on to a more generic security policies.

The description above does not cover one important aspect. Per application solutions try to affect only a very specific part of the system, but they must interact with the entire system. That often leads to a situation in which the tool has potentially very simple and straightforward use cases, but due to interactions with the rest of the system using it correctly becomes a very difficult task.

3.3.1. Setuid/setgid

These tools allow subject to temporary (or permanently) change its uid/gid, effectively changing the subject identity in the traditional DAC system (as described in [7]). This can be leveraged both to gain or drop some of the privileges.

Dropping privileges is needed in a situation when a privileged subject wants to follow the least privilege rule. If one of the actions that subject wants to perform require no special privileges, the subject can drop them before starting the action and regain them when the action is finished. For example a process updating the system may use setuid to drop administrative privileges for the time of downloading required files from the Internet.

In the world of traditional Unix DAC setuid is also necessary to allow non-root users to perform many operations — for example *ping* requires listening on the network interface for control packets, which requires administrative privileges. Another common use case is to allow user (not administrator) to perform system/application update. In other words the tool was needed to cover for the lack of flexibility in traditional Unix DAC.

There are two ways of using setuid/setgid mechanism:

- invoking *setresuid/setresgid* system call,
- executing a file which has setuid/setgid permission bit set.

The first action allows to drop/regain privileges. *Setresuid* takes three arguments euid, ruid and suid. They are integer values that will be assigned to effective uid, real uid and saved uid of the subject. If any of them has the value of -1, the system call will preserve the current value of the given uid. If the call is made by a subject with administrative privileges (one possessing CAP_SETUID) the arguments can be arbitrary. If the call is made by a subject without CAP_SETUID capability, all the values of the arguments must be equal to any of the effective, real or saved uid. The same semantics applies to setresgid() system call.

In order to gain privileges through setuid mechanism subject has to execute a file which has special permission flags set (to be precise setuid/setgid) flags. The process created that way will have effective uid of the file owner (most often of an administrator) but real uid of its parent, which is the subject. Thanks to that, subject can delegate privileged work to its child. To send data to the child, subject may use execution arguments or the environment. This design is secure in the sense that even if subject is subverted he must persuade the executable child to perform illegal operation. That means the security is guaranteed by the correctness of the setuid/setgid executable. If it is always able to correctly determine if the subject should be allowed to perform the requested action, than security is preserved. One can notice that here the responsibility for managing security shifts from the operating system to a single executable. That is why all the setuid applications must be written very carefully and are potential threat to system integrity.

Even though the setuid seems to be simple and effective mechanism it can be a source of confusion and dangerous security vulnerabilities. This is illustrated by the example of a sendmail bug — sendmail is a commonly used mail transfer agent. The main executable had a setuid permission flag set in order to allow users to send data to the global system-wide mail queue. In order to minimize the risk, sendmail would drop all the privileges as soon as possible. The privilege dropping was done through a call *setuid*(*getuid*()) system call. This call should set all effective, real and saved user IDs to the one returned by the *getuid*(). Function *getuid*() returns the uid of the user who executed the file. This seemed to be a simple and straightforward mechanism with no place for error. However, the semantics of the *setuid*() says that if the call is not allowed due to lack of permissions (no CAP_SETUID) it simply changes the effective user id and leaves the saved user id unchanged. In Linux kernel prior to 2.2.16 it was possible to remove CAP_SETUID while preserving the effective user id of administrator. The result of those two facts was a possibility that the attacker could run sendmail with full privileges.

3.3.2. Chroot

The tool is used to limit access of a jailed subject to a specific part of the file system. This has very powerful consequences, especially when it is combined with DAC. For example, one can permit a web server to modify user's *public_html* folder, but thanks to chroot limit the daemon will not be able to access any other user's files.

The call can only be made by a subject with administrative privileges (namely with CAP_CHROOT). When called, *chroot*(path) changes the resolution of the paths for the subject and all its descendants so that the path argument is treated as the root directory (/). However, the current working directory remains unchanged. Furthermore, the effect of the call can be reverted by a subject which possesses CAP_CHROOT capability.

As a consequence it is difficult to use chroot system call to set a permanent filesystem jail for a process. It can easily be used to restrict possible effects of software bugs, but it is not so useful for preventing malicious attack — the exact methods of breaking out of jail can be found in [5, 4].

3.3.3. Capabilities

Capabilities were introduced into Linux to allow finer grained separation of administrative privileges (the theoretical background for capabilities is covered by [2]). They are special flags attached to each subject. Each flag is a binary value defining whether the subject should be allowed to perform specific action.

Each subject possesses three sets of capabilities: effective, permitted and inheritable. The first one is used by the system to determine if the subject should be granted a permission to execute an action. The third one is the set of capabilities preserved across execv system call. The second is a

bound on the two other sets. If a capability is not in the permitted set, it cannot appear neither in inheritable nor in effective set. Furthermore, a subject is only allowed to remove capabilities from the permitted set. Any descendant of the subject is then limited to the capabilities in subject's permitted set. The only way to escape this limit is to execute a file with setuid flags.

Apart from that there also exists a capabilities bounding set. This set was originally global, but in newer kernels it is also per subject. This puts limit on all capabilities — even when a subject executes a file with setuid flag it cannot get a capability that is not in the bounding set.

In order to provide compatibility with the traditional Unix semantics of setuid/setgid capabilities sets are modified whenever the user id changes. The following rules apply:

- if all ids change to non-root (effective, real and saved) than the permitted and effective capabilities sets are cleared,
- if effective user id changes to non-root from root, effective set if cleared,
- if user id changes from non-root to root than permitted set is copied into effective set,
- if file system user id changes from root to non-root than CAP_CHOWN, CAP_DAC_OVERRIDE, CAP_DAC_READ_SEARCH, CAP_FOWNER, CAP_FSETID are cleared from the effective set.

Capabilities are a useful mechanism of limiting processes performing administrative tasks. They can prevent harmful operations performed as a result of software bugs in privileged subjects. However, they are not effective against malicious attackers. Capabilities are simply designed to divide administrative privileges. Almost all of them are equivalent to full root access to the system, when used properly. I will give few solid examples below:

- CAP_CHROOT — it allows the subject to use chroot system call. Subject can use it to change '/' to an arbitrary directory in which */lib/linux-ld.so.2* is maliciously substituted and later execute some setuid binary. The binary will link to the malicious library and as a result may execute attacker's code,
- CAP_MOUNT — allows mounting file systems. Subject may use it to mount read-write kernel memory.

3.4. Summary

The chapter showed a wide variety of solutions that are already available. As was clear from their analysis, they all lack either flexibility or simplicity. The next chapter will try to define a set of requirements that a sandboxing solution must posses in order to be powerful enough to satisfy most users, while still being relatively simple to use.

Chapter 4

Requirements analysis

In this chapter we will discuss functional and design requirements. Specific demands on providing sufficient security will be investigated in the first section. Design, described in the second section, has to answer the needs of programmers, users, and moreover has to make the code of crucial importance as simple as possible (to avoid introducing new bugs).

4.1. Functional requirements

There exist many systems of resource restriction in GNU/Linux. Primitive ones (such as capabilities) do not provide sufficient security, while complex (such as SELinux) are inconvenient for non-administrative use. End-user applications require simple interface with reasonable functionality. Systems overloaded with features cannot provide security in the area where settings depend on the user.

The main goal is the reduction of the immanent risk of SUID binaries, external device and filesystem access. It is also important to provide protection against DoS by fork-bombs or extensive memory/CPU usage. The list of minimal requirements for this purpose is as follows:

Process It is required that the process can switch to arbitrary uid and any number of given gids.

Environment of execution It is required that the process can switch to a specified environment or to a given subset of the original environment.

Descriptors It is required that the process can close all but the specified descriptors.

External devices It is required that the process can restrict its access to external devices. One set of external devices (pairs of major and minor numbers) can be specified for read-only access, one for read-write access, and one for mknod.

Virtual filesystem access It is required that the process can change its root directory. Moreover the new directory can be set up by the following operations:

1. binding all directories from the root of the filesystem (/) as read-write,

2. binding all directories from the root of the filesystem (/) as read-only,

3. binding only specified directories as read-only/read-write/empty,

4. binding specified directories as read-only/read-write and binding the remaining directories as read-only/read-write.

Binding a directory as empty will result in creating a new temporary directory. This functionality is useful for any kind of */tmp* directories.

CPU/Memory It is required that the process can restrict:

1. (CORE) maximal size of core file that can be created,
2. (CPU) available CPU time,
3. (FSIZE) maximal size of file that can be created,
4. (NICE) minimal nice number that the process can set,
5. (NOFILE) number of descriptors that can be opened,
6. (NPROC) number of processes that can be created,
7. (AS) size of memory that can be allocated

for each process created, and total CPU/memory usage of the group of processes forked of the main program.

PID namespace It is required that the process could be placed in a separate pid namespace, so that the inter-process communication is highly reduced.

Network namespace Process can be exploited through networking support. Providing secure network sandboxing is, however, very complex and bug-prone task. We have decided not to include this layer of isolation into our project mostly because of the potential risk of creating new security vulnerabilities.

4.2. Design requirements

A three-layer architecture is required. Kernel level support is necessary and consists of existing mechanisms such as rlimits, cgroups and VFS directory binding. Second layer (the library) is the fundamental component of the system. It is the library that offers simple and unified API for restricting process access to the resources. Third layer (the supervisor process) provides interface for the end-user. It has to check the consistency of settings and implement correct sandboxing techniques.

4.2.1. Kernel level

There is a large variety of kernel mechanisms providing security related functionality as kernel developers used many different approaches to supply userspace programs with convenient tools. The most well-established ones are DAC, ACL, SUID flags, and chroot, which are necessary for security but certainly not sufficient. Unfortunately there is no consistent extension to these foundations. There are, however, tools for a variety of security models that can be used by programmers.

Modifying the kernel gives a lot of flexibility and makes the solutions more powerful. The patches, on the other hand, are difficult to maintain and make the whole system very unfriendly for the user. It is therefore required that the sandbox uses already existing kernel mechanisms.

SECCOMP SECCOMP is an extension that allows restricting a thread to a small number of syscalls (read, write, sigreturn, exit) — thorough overview is available in [9]. The restricted thread is free to make any of these calls directly to the kernel. In case of other calls it has to turn to a trusted helper application, which makes the call on its behalf.
Advantages of SECCOMP:

- it is possible to decide whether to allow the syscall that the sandboxed program requests in real time,
- it is possible to handle exceptions (when a syscall request is denied).

Disadvantages of SECCOMP:

- it is necessary to adjust the code of a sandboxed program to the demands of SECCOMP,
- signal support needs a lot of work,
- in case of almost any software that performs I/O operations the limitations of SECCOMP are too prohibitive. The software needs to use an open file descriptor as a communication channel to a secure supervisor which could perform actions on behalf of untrusted code. This means that apart from adjusting the code of application one needs to write a secure supervisor fitted for it.

Ptrace Though created for a different purpose, ptrace can be used to control child process similarly to SECCOMP.
Advantages of ptrace:

- strict control over the sandboxed program resembles results that can be achieved by interpreting the code (in this case no adjustments to the code of a sandboxed program are required),
- it is possible to decide whether to allow the syscall that the sandboxed program requests in real time, for any syscall.

Disadvantages of ptrace basically come from the fact that writing a secure supervisor is extremely difficult. The are many reasons for this:

- exception handling is very complex — POSIX specification leaves many possible outcomes as undefined,
- kernel support for ptrace mechanism has much lower quality than most of the kernel code - it has many errors, some of them remaining unfixed for years, which gives little hope the they will be ever fixed,
- support for forking/threading is non existent — ptrace requires redefining process semantics, which is equivalent to requiring a complete rewrite the software.

Rlimit Supplementary to tracing Rlimit can be used to restrict resource usage by a sandboxed program.
Advantages of Rlimit:

- straightforward API,
- allows to restrict the number of descriptors the process can use.

Disadvantages of Rlimit:

- only very strict limits provide security,
- no exception handling is possible.

Cgroups and Namespaces Isolation from other processes can be done by tracing, but there exists a mechanism dedicated to it. Cgroups can be used to restrict total resource usage for a group of processes, prioritize it or separate it from other processes in the system.
Advantages of Cgroups:

- Cgroups limits are very convenient for sandboxed processes that are allowed to fork,
- it is possible to isolate groups of processes in separate PID namespaces, network namespaces, VFS namespaces, and IPC namespaces - one syscall can remove a lot of threat areas,
- these are kernel level isolation mechanisms, so many problems associated with userspace syscall filtering are solved (no risk of race conditions, thread safety, resistance against exceptions).

Disadvantages of Cgroups:

- the tool is relatively new and as such not very well tested and understood,
- tracing is more flexible,
- they require a relatively recent kernel.

4.2.2. C Library

It is required to choose a subset of available kernel mechanisms in order to create a consistent API for userspace programmers, featuring both simplicity and flexibility. It is not our intention to implement support for all existing mechanisms.

The interface has to be straightforward. Proposed model is based on the following scheme:

1. programmer creates an object which totally defines how the sandboxed process should be limited (by default no actions will be taken to secure the untrusted code),
2. programmer calls a number of functions that set different options of sandboxing in the created object,
3. programmer executes the untrusted program in one function call giving as an argument only the object he received from the library.

Our solution should implement this scheme. All parameters concerning sandboxing execution are kept in a *jail_param* structure:

```
struct jail_param {
        struct exec_param exec;
        struct std_param std_limit;
        struct vfs_param vfs_limit;
        struct rlimit_param r_limit;
        struct cgroup_param cgroup_limit;
};
```

This structure contains substructures which define parameters of various kernel-space mechanisms. Internal details of this structure are, transparent to the programmer, who sets the parameters using a set of functions:

```
void set_exec_path_param(struct jail_param *p, const char *path);
void add_exec_arg_param(struct jail_param *p, const char *path);
int set_env_type_param(struct jail_param *p, int type);
int add_env_param(struct jail_param *p, const char *name,
                  const char *value);
...
```

Afterwards the untrusted code can be executed using an implementation of

```
int start_jail(struct jail_param* params, int use_supervisor);
```

4.2.3. SUID supervisor

The system is mostly designed for programmers whose code executes untrusted applications (e.g. web browsers). Alternatively it can be used by an end-user to run an untrusted application directly (e.g. running a screensaver downloaded from the Internet). For this purpose a SUID supervisor application has to be created. It should receive all the parameters in arguments and execute the desired untrusted application.

The supervisor's task, however, is not restricted to parsing arguments and calling functions from the C library. There has to be a strong checking module that would not allow to exploit the SUID program itself (e.g. by running the sandboxed application with greater privileges).

SUID supervisor parameter passing schema should be sufficient for most use cases, but does not necessarily need to cover 100% of library's features. Our version of supervisor is developed mainly for testing and demonstrating possible usage scenarios. As such it only performs simple sanity checks on its arguments. In our implementation we propose the following usage pattern:

```
Usage: ./jail.py [argument [argument [...]] [--] command
```

While explaining the meaning of options of supervisor we will refer to the process of an untrusted application created by the supervisor as child. By the same convention the SUID tool will be referred to as parent. In the description below square bracket — "[]" — means that the value inside is optional. The '|' separates options of which only one can be selected. The parameters are:

–id=uid:gid1[,gid2,...,gidN] — (the parameter is required) run the command with effective, real and inherited user id = *uid* and group id = $gid1$ with possible secondary groups $gid2,\dots,gidN$,

–env_select=[NAME1[,NAME2[,...]] — copy the environment variables specified in the list $NAME_1,\dots,NAME_N$ into the environment of the child process, discard all the rest of the environment variables (for child process),

–fd_close_all — close all opened file descriptors for the child,

–fd_close_ex=[FD1[,FD2[,...]]] — close all opened file descriptors except for those specified on the list $FD_1,\dots,FD_N$,

–pid_namespace — contain the child process in a pid namespace,

–vfs_all=R|W — let the child process see the entire file system (the same view as the parent process), if option is equal to W then it will have normal access, if it is equal to R it will be read only (note */tmp* will always be writable),

–vfs_except=R|W[,src[,src[,...]]] — let the child process see entire file system except for directories specified in the list $src_1, \ldots, src_N$, the meaning of the values W, R is the same as in –vfs_all,

–vfs_select=[src:dst[:RE][,src:dst[:RE][,...]]] — let the child process see only the directories from the list where *src* stands for the original directory and *dst* for the path at which it will be mounted in the child's VFS; for each mount one may select options *R* — the mount will be read-only; *E* — the directory will be empty; furthermore one cannot specify */tmp* as target of the mount,

–mem_max=positive integer — limit the amount of memory available to the process and all of its children,

–device_allow=[a|b|c major:minor [r][w][,a|b|c minor:major [r][w]]] — let the child access only listed devices; each device is specified by three parameters separated by single spaces. First is one of: *a*- any type of device, *b*-block device, *c*- character device. Second one specifies *major* and *minor* number of the device, integer values are allowed as well as wildcast '$*$' standing for any number. Last one specifies access type: *w* — write only, *r* — read-only, *wr* — read-write,

–device_deny=[a|b|c major:minor [r][w][,a|b|c minor:major [r][w]]] — deny the child to access listed devices; the list specification is the same as for –device_allow. IMPORTANT: allow list has higher priority than deny list,

–limit_CPU=LIMIT_CPU — limit the available CPU time for the process (in seconds),

–limit_as=LIMIT_AS — limit the max size of virtual memory used by the process (in bytes),

–limit_nproc=LIMIT_NPROC — limit the number of processes that can be run with the same user id,

–limit_fsize=LIMIT_FSIZE — set the maximum size of a file that the child process can create,

–limit_core — forbid creating core dumps,

–limit_nofile — forbid opening any new file descriptors.

The command specifies the path to the executable file.

4.3. Adapting existing software

4.3.1. Overview

It is required that adapting existing software which executes untrusted application is made as easy as possible. Examples of such pieces of software are as follows: web servers (Apache, suEXEC), web browsers (firefox, chrome), shell servers (telnet, sshd).

Web servers run applications which theoretically could be trusted. Those applications, however, are usually exposed to a great number of people which makes attacks very likely. Moreover, programming languages like PHP don't make the difficult enough task of writing secure code any easier. One other reason why server code shouldn't be trusted is the fact that very often hundreds of users share it and each of them is potentialy introducing bugs. Successful attack to ones web page should not cause any harm to other users.

Web browsers require a lot of flexibility from the sandboxing tool. There can exist two very similar actions of which one is acceptable and the other is not. For an example uploading a photo selected by user is acceptable, while uploading all the other photos is not. Web browsers require real-time communication between the sandbox manager and the user.

Shell servers don't need sandboxing provided that all the code on the server is secure. In other cases sandboxing can greatly improve the security by disallowing some operations that are normally not performed by users. The other reasons to use sandboxing are CPU usage and memory restrictions.

4.3.2. Specific requirements

Executing untrusted applications is typically performed in the following sequence:

1. Security checks (validity of users in /etc/passwd, permissions to files/directories, current directory, user's home directory, etc.).
2. Hiding information (destroying everything from the virtual memory of the process that could be used by the attacker).
3. Dropping permissions (*setuid*() + *setgid*()).
4. Additional checks (checks that should not be performed as root).
5. Perform exec().

It is required that start_jail() could be performed in the place of exec() and all the steps 1-4 could be completed with minimal privileges. It is also required that some trivial security checks were done by the sandboxing library.

4.4. Summary

After this chapter the reader should have the knowledge about how the gap in the market can be characterized and what the specific functional requirements are. He should also be familiar with kernel mechanisms that can be used to provide necessary functionality. The main structure of the sandboxing solution and user's API were also covered in this chapter. Next one will elaborate on system architecture and the base of the implementation — kernel mechanisms.

Chapter 5

System architecture

5.1. Overview

By now it is clear that the three-layer structure is the optimal architecture for the sandboxing problem. It is flexible enough for programmers, friendly enough for the users, and secure enough thanks to its solid foundation — the Linux kernel. As it was argued, creating new patches is undesirable. Therefore existing mechanism have to be understood deeply. In this chapter we will present kernel features appropriate for sandboxing and select these used in our solution.

5.2. Available kernel mechanisms

In this section we will present Rlimits, capabilities, Cgroups, and namespaces. First two of those mechanisms are relatively old, and designed for administrative tasks rather than for sandboxing. To make use of them very strict limits have to be chosen. On the contrary, last two mechanisms are flexible, but complex as well.

5.2.1. Rlimits

Rlimits were designed to reduce the influence of bugs in software. Their main aim was to provide administrators with a simple tool for resource restrictions. An example of use-case is setting a memory limit for a process with memory leak bug. Kernel mechanism that killed such processes was a straightforward workaround.

5.2.1.1. Overview

Linux kernel provides a mechanism of resource limitation for each thread (see [6]). Each resource has associated limits (soft and hard) defined per thread in rlimit structure:

```
struct rlimit {
        rlim_t rlim_cur;
        rlim_t rlim_max;
};
```

Variable *rlim_max* defines the hard limit (which is also the maximum allowed value for *rlim_cur*). When *rlim_cur* is reached usually the process receives a signal; after reaching hard limit it is terminated.

The array of *rlimit* structures (*rlim*) is an element of *signal_struct,* which defines signal handlers for a tasks. Each thread has a pointer to *signal_struct* in its *task_struct*. On the other side each resource has a corresponding integer number which is the index in the *rlim* array.

```
struct task_struct {
        ...
        /* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;
        ...
};

struct signal_struct {
        ...
        struct rlimit rlim[RLIM_NLIMITS];
        ...
};
```

System calls *setrlimit* and *getrlimit* can be used to modify/check soft limit.

```
int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);
```

To use a resource without any limits *rlim_cur* must be set to RLIM_INFINITY.

5.2.1.2. Resources

Depending on the index in *rlim* tables the limit values have different meanings. Resources are represented as integer constants with prefix *RLIMIT_*, and limits for them are as follows:

RLIMIT_AS affects calls to brk, mmap and mremap. These will fail with ENOMEM error when exceeding the limit. Automatic stack expansion exceeding this limit will generate SIGSEGV. Below an example of RLIMIT_AS check can be found.

```
unsigned long do_brk(unsigned long addr, unsigned long len) {
        ...
        if (!may_expand_vm(mm, len >> PAGE_SHIFT))
                return -ENOMEM;
        ...
}

int may_expand_vm(struct mm_struct *mm, unsigned long npages) {
        unsigned long cur = mm->total_vm;        /* pages */
        unsigned long lim;
        lim = current->signal->rlim[RLIMIT_AS].rlim_cur >> PAGE_SHIFT;
        if (cur + npages > lim)
                return 0;
        return 1;
}
```

RLIMIT_CORE defines the maximum size of the core file. When the limit is set to 0 no core file can be created.

RLIMIT_CPU defines the maximum CPU time the process can get. After reaching soft limit the process receives SIGXCPU signal each second until the hard limit is reached. This signal can be handled, otherwise the process is terminated.

```
unsigned long psecs = cputime_to_secs(ptime);
unsigned long hard = ACCESS_ONCE(sig->rlim[RLIMIT_CPU].rlim_max);
cputime_t x;
if (psecs >= hard) {
        /* At the hard limit, we just die.
         * No need to calculate anything else now. */
        __group_send_sig_info(SIGKILL, SEND_SIG_PRIV, tsk);
        return;
}
```

RLIMIT_DATA specifies the maximum size of the process' data segment, which consists of initialized/uninitialized data and the heap. It affects brk and sbrk syscalls. The following check originates from brk function:

```
rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
if (rlim < RLIM_INFINITY && (brk - mm->start_brk)
        + (mm->end_data - mm->start_data) > rlim)
        goto out;
```

RLIMIT_FSIZE specifies the maximum size of files that the process may create. This affects many calls including *write* and *truncate*. The following function is used to check the limitations:

```
inline int generic_write_checks(struct file *file, loff_t *pos,
                                size_t *count, int isblk) {
        ...
        unsigned long limit = rlimit(RLIMIT_FSIZE);
        if (!isblk) {
                ...
                if (limit != RLIM_INFINITY) {
                        if (*pos >= limit) {
                                send_sig(SIGXFSZ, current, 0);
                                return -EFBIG;
                        }
                        if (*count > limit - (typeof(limit))*pos) {
                                *count = limit - (typeof(limit))*pos;
                        }
        .......
}
```

RLIMIT_LOCKS has no effect in kernels newer than 2.6.0.

RLIMIT_MEMLOCK specifies the maximum number of bytes of memory that may be locked into RAM. The actual limit is rounded down to the nearest multiple of the system page size. This limitation affects *mlock*, *mlockall*, *mmap, shmctl* kernel functions. Shared memory locks are counted

separately from per-process memory locks. The process may lock memory up to *rlim_cur* bytes in both categories.

RLIMIT_MSGQUEUE specifies the maximum number of bytes that can be allocated for POSIX message queues for the user that called the process. This limitation affects *mq_open* kernel function. The number of bytes is calculated as a sum of

$$attr.mq_maxmsg \cdot (\text{sizeof}(\text{struct } msg_msg*) + attr.mq_msgsize)$$

for all messages, where *attr* is the structure specified as the fourth argument of *mq_open*. As a consequence of this formula the user cannot create an unlimited number of empty messages.

RLIMIT_NICE specifies the maximal nice value that the process can be raised to. The actual maximal nice value is calculated as

$$20 - rlim_cur.$$

This awkwardness is caused by the fact that negative *rlim_cur* values are meaningful (e.g. $RLIM_INFINITY = -1$). Kernel function *can_nice* is used to check the RLIMIT_NICE before renicing.

```
int can_nice(const struct task_struct *p, const int nice) {
        /* convert nice value [19,-20] to rlimit style to [1,40] */
        int nice_rlim = 20 - nice;
        return (nice_rlim <= task_rlimit(p, RLIMIT_NICE) ||
                capable(CAP_SYS_NICE));
}
```

RLIMIT_NOFILE specifies the value of the maximum file descriptor number that can be opened by this process. The actual maximal number is calculated as

$$rlim_cur - 1.$$

RLIMIT_NPROC specifies the maximum number of threads that can be created by the user that called this process. This limit affects *fork,* which returns *EAGAIN* error when encounters it. Certain capabilities may however waive this effect:

```
static struct task_struct *copy_process (unsigned long clone_flags ,
    ....
        unsigned long stack_start ,  struct pt_regs *regs ,
        unsigned long stack_size , int __user *child_tidptr ,
        struct pid *pid , int trace ) {
        if (atomic_read(&p->real_cred->user->processes) >=
                task_rlimit(p, RLIMIT_NPROC)) {
                if (!capable(CAP_SYS_ADMIN) &&
            !capable(CAP_SYS_RESOURCE) &&
                        p->real_cred->user != INIT_USER)
                                goto bad_fork_free;
                }
        ....
}
```

RLIMIT_RSS has no effect in kernels newer than 2.4.30.

RLIMIT_RTPRIO specifies the maximal value of the real-time priority that may be set for this process. This limit affects *sched_setscheduler* and *sched_setparam.*

RLIMIT_RTTIME specifies the number of microseconds of CPU time that a process scheduled under real-time policy may consume without making a blocking system call. Each time the process makes such a call CPU timer is reset to zero. It does not, however, reset when the process is preempted, its time slice expires or it calls *sched_yield*.

RLIMIT_SIGPENDING specifies the maximal number of signals that may be queued for the user who called the process. This limitation affects only *sigqueue* kernel function, so it is always possible to use kill to queue one instance of any of the signals that are not already queued to the process. When the *RLIMIT_SIGPENDING* check fails the information about dropped signal is printed by *printk*.

```
static struct sigqueue * __sigqueue_alloc(int sig,
                                  struct task_struct *t,
                                  gfp_t flags,
                                  int override_rlimit) {
        ...
        if (override_rlimit ||
                atomic_read(&user->sigpending) <=
                    task_rlimit(t, RLIMIT_SIGPENDING)) {
                q = kmem_cache_alloc(sigqueue_cachep, flags);
        } else {
                print_dropped_signal(sig);
        }
        ...
}
```

RLIMIT_STACK specifies the maximum size of the process stack in bytes (memory space used for command-line parameters and environment variables are also taken into consideration). Signal *SIGSEGV* is generated when the limit is exceeded.

5.2.2. Cgroups

The abbreviation Cgroups stands for control groups (overview of the mechanism available in [8]). Basically, a control group is a set of processes with specific limits/usage counts assigned to the set. The main purposes of cgroups are:

- accounting — measure how much of given resource is used,
- isolation — provide different views of the resources available in the system to different processes,
- limitation — assert that process will not use too much of given resource,
- prioritization — make sure that some processes have higher/lower priority in accessing certain resources.

The cgroups can be hierarchical — a cgroup can be nested inside other cgroups. In that case the limitations/counts of all ancestor cgroups apply also to the nested cgroups. Cgroup mechanism is divided into a few subsystems. Each subsystem provides tools to limit/measure usage of different kind of resource. Currently the following subsystems exists:

- blkio — used to allow prioritization and limitations of I/O operations on block devices,
- cpuacct — used for measuring the CPU usage (as number of nanoseconds of CPU time),
- cpusets — allows to assign specific CPU and memory nodes to a process — the processes will not be migrated outside these nodes,
- devices — used to define which devices can be accessed (possible actions are: read/write/mknod) by processes,
- freezer — allows to stop and start a group of processes in a single, atomic operation,
- memory — measures the usage of memory and allows to set thresholds, so that the processes cannot exceed their limits.

5.2.3. Namespaces

In Linux kernel, namespaces refer to a mechanism which allows partitioning available resources, so that processes can only see and access these which belong to their partition. Each namespace defines what should be discoverable from within — any other resource does not exist from the perspective of a process belonging to the namespace. The mechanism provides a very simple and straightforward way of dividing the system into domains whose interactions are limited. The following namespaces are available:

- mount — the processes in the namespace have their own view of the file system; mount operations performed in the namespace do not need to be visible to other processes and vice versa ([1] offers an extensive tutorial on using mount namespaces),
- pid — process in the namespace cannot see pids and process information of processes from outside the namespace,
- network — isolates network devices, iptables rules, routing rules; allows each namespace to treat its network traffic differently,
- ipc — separates SysV inter process communication mechanisms, so that processes can only use them for communication within the namespace,
- uts — allows changing the hostname only within the namespace.

5.2.4. Capabilities

Since version 2.2 Linux kernel divides superuser privileges into groups called capabilities (thorough explanation of the mechanism is available in [7]) . This (in theory) allows processes to obtain some, but not all, of the privileges. In reality, however, many of these capabilities are equivalent to all superuser privileges.

5.2.4.1. List of capabilities

CAP_AUDIT_CONTROL Allows enabling and disabling kernel auditing, changing auditing filter rules and retrieving auditing status and filtering rules.

CAP_AUDIT_WRITE Allows writing records to kernel auditing log.

CAP_CHOWN Allows chown operations on arbitrary files.

CAP_DAC_OVERRIDE Bypasses file read, write, and execute permission checks.

CAP_DAC_READ_SEARCH Bypasses file read permission checks and directory read and execute permission checks.

CAP_FOWNER Bypasses all restrictions about allowed operations on files, where file owner ID must be equal to uid.

CAP_FSETID Allows modifying files and preserving their SUID and SGID bits. Allows setting the sgid bit for a file whose gid does not match the file system or any of the supplementary gids of the calling process.

CAP_IPC_LOCK Allows memory locking (*mlock*, *mlockall*, *mmap*, *shmctl*).

CAP_IPC_OWNER Bypasses permission checks for operations on System V IPC objects.

CAP_KILL Bypasses permission checks for sending signals.

CAP_LEASE Establishes leases on arbitrary files.

CAP_LINUX_IMMUTABLE Allows setting the FS_APPEND_FL and FS_IMMUTABLE_FL i-node flags.

CAP_MKNOD Allows creating special files using mknod.

CAP_NET_ADMIN Allows performing various network-related operations (e.g., setting privileged socket options, enabling multicasting, interface configuration, modifying routing tables).

CAP_NET_BIND_SERVICE Allows binding a socket to ports under 1024.

CAP_SETGID Allows arbitrary manipulations of process GIDs and supplementary GID list.

CAP_SETFCAP Allows setting file capabilities.

CAP_SETPCAP Allows transferring and removing capabilities from the permitted set to and from any pid.

CAP_SETUID Allows making arbitrary manipulations of process UIDs (*setuid*, *setreuid*, *setresuid*, *setfsuid*).

CAP_SYS_ADMIN Allows many administration tasks (setting domainname, hostname, configuring *printk* behaviour, mounting and umounting, locking/unlocking shared memory segments, turning swap on/off, setting readahead and flushing buffers on block devices, turning DMA on/off, administrating RAID devices, accessing nvram device, setting up serial ports, setting encryption on loopback filesystem, etc.).

CAP_SYS_BOOT Allows calling *reboot* and *kexec_load* functions.

CAP_SYS_CHROOT Allows calling *chroot* function.

CAP_SYS_MODULE Allows loading and unloading kernel modules.

CAP_SYS_NICE Allows raising priority and setting priority on other users' processes; allows using roundrobin scheduling on owned processes.

CAP_SYS_PACCT Allows calling *acct*.

CAP_SYS_PTRACE Allows tracing arbitrary processes using *ptrace*.

CAP_SYS_RAWIO Allows performing I/O port operations (*iopl*, *ioperm*) and accessing */proc/kcore*.

CAP_SYS_RESOURCE Overrides quota limits, reserved space on ext2 filesystems, resource limits. Allows modifying journaling mode on ext3 filesystems, setting resource limits.

CAP_SYS_TIME Allows setting the system clock (*settimeofday*, *stime*, *adjtimex*) and the hardware clock.

CAP_SYS_TTY_CONFIG Allows calling *vhangup*.

CAP_SYSLOG Allows performing privileged syslog operations.

5.2.4.2. Examples of capability equivalence

Capabilities are designed to reduce the risk of unintentionally using special privileges in a bad way. However, they are not intended to be used with untrusted code. When considering arbitrary code execution many capabilities are equivalent to full superuser privileges. A few examples are presented below:

Equivalence of CAP_SETUID with full superuser privileges is trivial. The process may set SUID-root permissions to /bin/bash and execute it.

The case with CAP_SYS_MODULE is also trivial, but requires much more work for the attacker. The easiest scenario is preparing a kernel module that changes euid of certain processes and then loading it.

Having CAP_SYS_ADMIN an attacker can mount a prepared file containing a filesystem with SUID-root binary to some directory, and execute the binary.

Let us now consider a less trivial case, namely CAP_SYS_CHROOT, which seems harmless. In fact the attacker can prepare a directory containing his own libc and a SUID-root binary (created as a

hardlink). When executed under chroot the binary will use functions (such as malloc) from the local libc, prepared by the attacker. These functions will be executed with superuser privileges, and will not be restricted by any capabilities.

5.2.4.3. Capability bounding

Capability bounding set is a security mechanism designed to limit the set of capabilities that can be gained during exec(). A process cannot inherit capabilities which don't belong to capability bounding set, however it always inherits the capability bounding set. Having CAP_SETPCAP a process can remove a capability from its capability bounding set.

5.2.4.4. Security bits

Mechanism of security bits is designed to disable special handling of capabilities for uid=0 case.

SECBIT_KEEP_CAPS Allows a process to keep its capabilities after switching to uid>0.

SECBIT_NO_SETUID_FIXUP Stops kernel from adjusting capability sets when switching between uid zero and nonzero.

SECBIT_NOROOT Stops kernel from granting capabilities when a setuid program is executed.

5.3. Mechanisms used in sandboxing library

Current implementation of our sandboxing library uses the following kernel mechanism:

1. Rlimits:
    (a) RLIMIT_CORE,
    (b) RLIMIT_CPU,
    (c) RLIMIT_FSIZE,
    (d) RLIMIT_NICE,
    (e) RLIMIT_NOFILE,
    (f) RLIMIT_NPROC,
    (g) RLIMIT_AS.
2. Cgroups:
    (a) cpuacct,
    (b) memory,
    (c) devices.
3. Namespaces:
    (a) pid,
    (b) mount,

4. Capabilities:
    (a) security bits,
    (b) dropping capability bounding set.

5.4. Summary

By now the reader should be familiar with the foundation of our sandboxing solution and understand possibilities that the kernel mechanisms give. Next chapter will describe how those possibilities were put into practice.

Chapter 6

Implementation

6.1. Overview

This chapter will describe the actual implementation of the tools built by the authors. Its goal is to show how they can be applied by programmers to add security layer to their software. Furthermore, it will cover some of the problems authors have encountered while creating their solution; it also contains excerpts of source code with extensive comments about their meaning. The chapter is divided into parts describing different components of the solution, each part focuses on a different functionality.

6.2. Public API

Public API exposed by the library (as seen from C language header):

```
161 struct jail_param* create_jail_param();
162 void print_jail_param(struct jail_param *p);
163
164 void set_exec_path_param(struct jail_param *p, const char *path);
165 void add_exec_arg_param(struct jail_param *p, const char *path);
166 int set_env_type_param(struct jail_param *p, int type);
167 int add_env_param(struct jail_param *p, const char *name,
168                   const char *value);
169
170 int add_rlimit_param(struct jail_param *p, int resource,
                        rlim_t max);
171
172 void set_id_param(struct jail_param *p, uid_t uid, gid_t gid);
173
174 void set_pid_namespace_param(struct jail_param *p, int set_true);
175
176 void add_gid_param(struct jail_param *p, gid_t gid);
177
178 int set_mount_type_param(struct jail_param *p, int type);
179
180 int add_mount_param(struct jail_param *p, const char *src,
181                    const char *dest, int flags);
182
```

```
183 int set_fd_type_param(struct jail_param *p, int type);
184 int add_fd_param(struct jail_param *p, int fd);
185
186 int set_cgroup_mem(struct jail_param* p, long long val);
187 int set_cgroup_CPU(struct jail_param* p, long long val);
188 int add_cgroup_device(struct jail_param* p, int allow,
                              char* acc);
189
190 int start_jail(struct jail_param*, int);
```

The library is built around jail_param structure. All functions use a pointer to this structure. Thanks to that, its implementation can be changed without breaking the public API — users should never try to directly modify the structure fields. The entire external API of the library was build around the idea that it must be simple and easy to use even from non C/C++ languages. All functions take as an argument an opaque pointer and some other arguments which can only be integers or strings. Almost all higher level programming languages provide bindings to C (as almost all system libraries follow executable format consistent with compiled C). As integers, strings and pointers are usually well supported by those bindings, the task of porting the library API to a new language should be straightforward. The API can be split into two parts:

- functions modifying sandbox's settings (modifying the fields of jail_param),
- sandbox execution.

All functions except for the *start_jail* fall into the first category. These are described here one by one:

create_jail_param() — returns a pointer to the region of memory containing the jail_param structure. It is guaranteed that the pointer can be safely freed by calling *free*() function. In case of error returns NULL.

set_exec_path_param(p, path) — sets the path to the binary to be executed inside sandbox to the value of path.

add_exec_arg_param(p, arg) — adds a new argument to the binary to be executed. Arguments will be given in the order in which they were added.

set_env_type_param(p, type) — defines how the environment variables given to add_env_param should be understood. Possible values are: ENV_NOACTION (ignore the values, don't change the environment variables at all), ENV_SELECT (clear all current environment variables, add those which were explicitly added by add_env_param).

add_env_param(p, name, value) — adds an entry to the list of environment variables in the form of "name=value", for example name="PATH", value="*/usr/bin*".

add_rlimit_param(o, res, max) — sets a resource limit using rlimit interface. The res argument is one of RLIMIT_CORE, RLIMIT_CPU, RLIMIT_FSIZE, RLIMIT_NOFILE, RLIMIT_NPROC, RLIMIT_AS — the meanings of these flags ware explained in previous chapters. The max argument specifies the maximum value of resource usage allowed for the process (it will be both soft and hard limit). The value of *max* cannot exceed the limit set for the process which creates the sandbox.

set_id_param(p, uid, gid) — sets the identity of the sandboxed process to user id = uid (all real, effective and saved) and group id = gid (again all of them).

set_pid_namespace(p, val) — if val is logical true in C (any value different than 0) then the process will be executed in a separate pid namespace.

add_gid_param(p, gid) — adds an additional group to the list of supplementary groups of the process.

set_mount_type(p, type) — defines how VFS sandboxing will be done. Possible values are: BIND_ROOT (sandbox will see the entire file system), BIND_ROOTRO (sandbox will see the entire file system but in read-only mode), BIND_SELECT (sandbox will see only directories specified with add_mount_param), BIND_EXCEPT (sandbox will see the entire file system except for directories specified with add_mount_param), BIND_EXCEPTRO (same as BIND_EXCEPT, only the file system will be seen as read-only).

add_mount_param(p, source, dest, type) — adds a mount point to the list of mount points for the sandbox. It only makes sense if first, function *set_mount_type* was called earlier with BIND_SELECT, BIND_EXCEPT or BIND_EXCEPTRO argument. The arguments are as follows: source — the directory to be mounted inside the sandbox, dest — path inside the sandbox where the directory will be visible, type — specifies one of the mount options: BIND_RO (read-only access), BIND_EMPTY (the directory will be visible but with no content) or 0 meaning default behaviour.

set_fd_type_param(p, type) — defines how to treat file descriptors which are open at the moment of execution. Possible values: FD_CLOSE_ALL (close all file descriptors), FD_CLOSE_EX (close all except for those added by *add_fd_param*), FD_NOACTION (do nothing to file descriptors).

add_fd_param(p, fd) — add a file descriptor to the list of those which should not be closed when using
FD_CLOSE_EX flag in set_fd_type_param.

set_cgroup_mem(p, limit) — sets the maximal memory + swap usage by the sandbox. The value is interpreted as number of kilobytes and is rounded to full page sizes (i.e 4096 bytes on x86/x86_64 architecture).

*set_cgroup_*cpu(p, limit) — sets a limit on maximum CPU usage by the sandbox, the limit value should be specified in milliseconds.

add_cgroup_device(p, allow, device) — allows or restricts access to a device file inside the sandbox. Second argument specifies whether the access should be allowed (logical true) or denied (logical false). The string defining the device and access type has the following format: “D M:M ABB”, where D::a|b|c, M::integer | *, A::m|r|w, B::A|ε (using standard BNF notation). The meaning of the above string is as follows: D — stands for device type (a — any kind, c — character, b — block device), M:M — major:minor number of the device or * (any number), ABB — type of access allowed/denied (m — mknod on device, r — read from device, w — write to device). The list of allowed devices has a priority over the list of denied device. That is if one adds a device to both lists (allowed and denied) the device will be allowed inside the sandbox.

All the above functions only modify the jail_param structure given as their first argument. They perform no system calls. When everything is set, the user calls start_jail function with a pointer to jail_param.

start_jail(p, super) — kick starts the sandbox according to the settings of the first argument. If the second argument is logical true the sandbox will be monitored by the supervisor process. Note that setting this value to false will automatically disable CPU usage limits.

6.3. C library implementation

The library was implemented in GNU99 C standard. The reason for deviating from ANSI C99 standard is that the code is not portable to other Unix operating systems anyway (cgroups and bind mounts are Linux only features) and thanks to the GNU C extensions, implementation of internal structures is cleaner and simpler.

In order to make the library as self contained as possible and portable to variety of Linux systems, we decided not to use any external libraries apart from libcap (which provides functions for manipulating capabilities) and libc. Libc usage is restricted to POSIX standardized functionality. The only key piece of code that is external, is the implementation of list data structure, however, it is based on preprocessor macros and is contained in a single header file included in the source code. Below we will present selected parts of our code and explain their meaning.

```
20 #define BAD_RETURN(call, bad, retval)   do {\
21                                   if((bad) == (call)) {    \
22                                    DLOG(STRINGIFY(call)    \
23                                    " has failed");         \
24                                    DERRNO;                 \
25                                    return (retval);        \
26                                  }                         \
27                                } while(0)
28
...
31 #define FATAL                    do {                  \
32                                      DERRNO;              \
33                                      _exit(1);            \
34                                    } while(0)
```

The two macros defined above are used throughout the code of the library so it is crucial to understand what they do. The first one (*BAD_RETURN*) checks the result of calling its first argument (call); if it is equal to the second argument (bad) it prints the error message contained in *errno* global variable and the information about location in code (DERRNO macro) and returns the third argument. It is essentially used to check the return status of a syscall and in case of fail print the apropriate information without any attempts of error-recovery. If the return value is different than the second argument then the macro is equivalent to simply calling the first argument. The FATAL macro is used to denote a critical condition (no hope for fixing) — it calls the _exit syscall and ends the process immediately, without even calling the exit handlers.

```
159 int start_jail(struct jail_param* params, int use_supervisor) {
160   pid_t child;
161   pid_t parent;
162   uid_t uid;
163   gid_t gid;
164   int message_fd;
165   int cgroup_wait_fd;
166   int fifo_fd;
```

```
167   unsigned int ev;
168   struct jail_paths *paths;
169   char **args;
170   char **env;
171
172   parent = getpid();
173
174   /**Clone with new file system namespace (CLONE_NEWNS) and
175    * possibly with pid namespace (CLONE_NEWPID)
176    */
177   child = primitive_fork(SIGCHLD, 1,
                             params->std_limit.pid_namespace, 1);
178
179   if (child == 0) {
180     /*child process*/
181     ///* Ignore hangup signal */
182     BAD_RETURN(ignore_signal(SIGHUP, NULL), -1, -1);
183
184
185     BAD_RETURN(close_fds(&(params->vfs_limit)), -1, -1);
186
187     paths = create_jail_paths();
188     BAD_RETURN(paths, NULL, -1);
189     BAD_RETURN(create_tmp_jail_dirs(paths), -1, -1);
190
191     if(use_supervisor) {
192     // this channel is used to tell child it can continue,
193     // supervisor has started and set all the handlers
194       message_fd = eventfd(0, 0);
195       cgroup_wait_fd = eventfd(0, 0);
196       BAD_RETURN(message_fd, -1, -1);
197
198       child = primitive_fork(SIGCHLD, 1, 0, 1);
199     }
200
201     if(child == 0) {
202       BAD_RETURN(unshare_vfs(), -1, -1);
203       BAD_RETURN(jail_cgroup(&(params->cgroup_limit), paths),
                   -1, -1);
204       BAD_RETURN(jail_vfs(&(params->vfs_limit), paths), -1, -1);
205
206       if (use_supervisor) {
207         //send supervisor a message that it can carry on -
208         //child in a proper cgroup
209         write_event(cgroup_wait_fd, 1);
210         close(cgroup_wait_fd);
211       }
212
213       if (params->std_limit.pid_namespace)
```

```
214           BAD_RETURN(remount_proc(), -1, -1);
215         BAD_RETURN(drop_setuid(), -1, -1);
216         BAD_RETURN(set_identity(&(params->std_limit)), -1, -1);
217         BAD_RETURN(drop_capabilities(), -1, -1);
218         BAD_RETURN(set_limits(&(params->r_limit)), -1, -1);
219 
220         if(use_supervisor) {
221         //wait for the supervisor to set the handlers for events
222           BAD_RETURN(read_event(message_fd, &ev), -1, -1);
223           close(message_fd);
224         }
225 
226         prepare_exec(&(params->exec), &args, &env);
227         exec_process(params->exec.path, args, env);
228       }
229 
230       else if(child > 0) {
231       //supervisor process
232 
233       //fifo used for talking to supervisor
234         fifo_fd = create_fifo(paths);
235         BAD_RETURN(fifo_fd, -1, -1);
236         BAD_RETURN(unshare_vfs(), -1, -1);
237         if(params->std_limit.pid_namespace)
238           BAD_RETURN(remount_proc(), -1, -1);
239 
240         BAD_RETURN(get_id(&uid, &gid), -1, -1);
241 
242         //become root - only root can control it
243         BAD_RETURN(set_id(0, 0, 0, NULL), -1, -1);
244 
245         //wait until child goes to right cgroup
246         BAD_RETURN(read_event(cgroup_wait_fd, &ev), -1, -1);
247         close(cgroup_wait_fd);
248 
249         //execute supervisor binary
250         exec_supervisor(paths, child, message_fd, fifo_fd,
251                         uid, gid, params->cgroup_limit.CPU_max);
252         FATAL;
253       }
254     } else if(child > 0) {
255       return child;
256     }
257 
258   return -1;
259 }
```

The *start_jail* function is the one that performs the kick start of the sandbox. All other externally exposed functions are only used for configuration purposes. It starts with forking a new process

(line 177). The created process will always be in a new file system namespace — its mounts will be invisible to the parent (and generally to all of the processes which are not his descendants). Also, depending on the settings it may be in a separate pid namespace (option specified in sys_clone system call — line 177). The calling process will simply get the pid of the created child — line 255.

Newly created child starts by setting a signal handler for the SIGHUP signal — it is send when the process is detached from the terminal device (182) — and closing file descriptors (185). Which descriptors should be left open is determined by the sandbox options. The only important thing to remember here is the fact that the process should be able to open additional file descriptors later on — they are used in sandbox setup when supervisor is in use. After line 185, the process should have at most maximum allowed number of descriptors less 3 opened.

Between lines 186 and 189 temporary directories, needed for VFS sandbox, are created — their exact hierarchy and usage is described in the next section.

The next step is required only if we actually use supervisor process (lines 191–199). In that case we will need two message channels — one for the supervisor to wait for the sandboxed child to put itself into the right cgroup, the other one for the child process to wait with executing the actual unsafe code until the supervisor starts and sets all the event handlers. These channels are implemented using simple eventfd file descriptors (lines 194-195). Such descriptors are simpler and use less memory than ordinary kernel pipes on one hand, and provide an adequate level of synchronization on the other. The last step is forking off the child process (198). In this moment the supervisor will go to line 230 and the untrusted child to line 201. If supervisor is not used all the synchronization is not required (however, one will have to remove some temporary files manually) and we carry on directly (without fork) to untrusted child code.

The sandboxed process starts its execution with entering into the new VFS namespace (202) — this step is unnecessary is the supervisor is disabled (as it has already been done), but because of the low cost of the call, it is performed anyway. In line 203 we add the process to a proper cgroup and set resource limits for the group according to the sandbox settings — more detailed explanation is included in the following section. Line 204 jails the process inside the VFS sandbox. As this call is very complex it will be explained in more detail later on.

As the child process puts itself into the proper group it sends a message to the supervisor to carry on (lines 206–211). If we want to use a pid namespace we have to remount */proc* directory as the old one contains entries for all the processes in the system (after remounting the namespace will ensure that */proc* contains only its members' entries) — lines 213–214.

In lines 215–219 the child, which was executed from setuid binary, drops the privileges it gained. Firstly, it makes sure that the OS will not grant the process any capabilities on SUID binary execution — this is implemented using the *prctl*(SET_SECURE_BITS) system call. Secondly, it sets its identity to the one defined in the settings. The third operation is crucial — the process drops all the capabilities it has. Even if the process is not being executed with euid= 0 it can still perform many dangerous operations once it possesses certain capabilities. What is more most capabilities are easily turned into full superuser access by a smart attacker. Therefore, to provide sufficient security all the capabilities should be disabled. Moreover, the call in line 217 sets the process capability bounding set to empty. This means that even if it executes a binary which has some capabilities in its file system flags they will be ignored. Lastly, the process sets resource usage limits. This is done after dropping all privileges, so this operation cannot increase the limits above the level of the parent process (the jail creator) — there is no need to check. The crucial feature of these 4 lines is that after executing them in that order, their effect is irreversible. From that moment on the process will not be able to gain any privilege, access, increase its limits above the ones set in (218).

In lines 220–224 the child process waits for the supervisor to get ready. When it gets the 'go-ahead' message, it prepares the environment variables and command line options and executes the untrusted code (226–227). At this point the sandbox must be air-tight — the assumption in the design

was that the executed code will be totally controlled by the attacker.

```
74 /** pivots the new root of the filesystem using bind mounting
75  *  be extra carefull, calling this without first using
76  *  unshare(CLONE_NEWNS) may cause you system to change "/"
77  */
78 int pivot_vfs(const char* abs_new_root, const char *abs_old_root,
79               const char *relative_old_root) {
80
81   if ((check_file_type(abs_new_root) != S_IFDIR) ||
82       (check_file_type(abs_old_root) != S_IFDIR))
83       BAD_RETURN(-1, -1, -1);
84
85   BAD_RETURN(chdir(abs_new_root), -1, -1);
86
87   BAD_RETURN(mark_mount(abs_old_root, MS_UNBINDABLE),
               -1, 1);
88
89   BAD_RETURN(syscall(__NR_pivot_root, ".", abs_old_root),
               -1, -1);
90
91   BAD_RETURN(mark_mount(relative_old_root, MS_PRIVATE | MS_REC),
               -1, -1);
92   BAD_RETURN(umount2(relative_old_root, MNT_DETACH), -1, -1);
93
94   return 0;
95 }
...
377 int close_fds(struct vfs_param *param) {
378   int i = 0;
379   int all = getdtablesize();
380   struct descriptor_list *dlist;
381
382   int flag = (param != NULL)?
                param->descriptors : FD_CLOSE_ALL;
383
384   if(flag == FD_NOACTION)
385     return 0;
386
387    for(i=0; i < all; ++i) {
388
389      /**simply check if the fd is in a list*/
390      if(flag == FD_CLOSE_EX)
391        list_for_each_entry(dlist, &(param->fds), siblings)
392        if(i == dlist->fd)
393          continue;
394
395      if(close(i) != 0 && errno != EBADF)
396       BAD_RETURN(-1, -1, -1);
```

```
397     }
398
399     return 0;
400  }
...
403  int jail_vfs(struct vfs_param *param, struct jail_paths *paths) {
404     int ro = 1;
405     struct bind_list *blist;
406     /**mark the new root (as well as its parent) as unbindable,
407      * this is required for the pivot_root to work*/
408     BAD_RETURN(mark_mount(JAIL_PARENT_DIR, MS_UNBINDABLE), -1, -1);
409     BAD_RETURN(mark_mount(JAIL_ROOT_DIR, MS_UNBINDABLE), -1, -1);
410
411     switch(param->bind_type) {
412        case BIND_ROOT:
413        case BIND_EXCEPT:
414           ro = 0;
415        case BIND_EXCEPTRO:
416        case BIND_ROOTRO:
417        /**mount the / at the temporary directory to which we
418         * will chroot later on */
419           BAD_RETURN(bind_mount(JAIL_ROOT_DIR,"/", MS_PRIVATE|MS_REC,
420                                    MS_PRIVATE | MS_REC, ro), -1, -1);
421
422           break;
423        case BIND_SELECT:
424        /**mounting directories requested by the user,
425         * if creates /tmp or anything below the next call
426         * will fail, generally user is not allwed to touch /tmp
427         */
428           list_for_each_entry(blist, &(param->binds), siblings)
429              BAD_RETURN(bind_in_jail(blist->destination, blist->source,
430                            JAIL_ROOT_DIR, blist->flags == BIND_OPT_RO),
431                         -1, -1);
432
433           BAD_RETURN(safe_create_jailed_dir("/tmp",
                                               JAIL_ROOT_DIR), -1, -1);
434           break;
435        default:
436           return -1;
437     }
438
439     /*provide private, per user '/tmp' dicrectory*/
440     BAD_RETURN(bind_mount(JAIL_TMP_DIR, paths->tmp,MS_PRIVATE|
441                              MS_REC, MS_PRIVATE | MS_REC, 0),
                -1, -1);
442
443     /** the chroot itself, point of no return*/
444     BAD_RETURN(pivot_vfs(JAIL_ROOT_DIR, JAIL_OLD_ROOT_ABS,
```

```
445                                JAIL_OLD_ROOT_REL),
446                   -1, -1);
447
448     /**now its time to umount the directories*/
449     if (param->bind_type == BIND_EXCEPTRO)
450       list_for_each_entry(blist, &(param->binds), siblings) {
451         BAD_RETURN(bind_mount(blist->source, JAIL_OLD_ROOT_REL,
452                                    MS_PRIVATE, MS_PRIVATE, 0), -1, -1);
453       }
454
455     return 0;
456 }
```

Once the temporary directories needed for the sandbox are created we need to jail the process inside them. Usually the easiest way to jump out of a VFS jail is by using a file descriptor that was left open by the parent. If the library user is sure that there are no dangerous file descriptors remaining open, nothing has to be done. However, the practice shows that a file left open is a common bug which does not demonstrate itself easily. This is the reason why we decided to provide the functionality of closing file descriptors for the user. This is done by the function *close_fds* (lines 377–400). It supports two modes of operation: close all file descriptors or close all except for those specified by the user. As there is no portable and easy way to close all opened file descriptors (one can read this data from */proc* pseudo file system, but that solution is both complex and error prone) — we simply implemented it by finding the maximal file descriptor allowed for the process (379) and trying to close all descriptors below it (395). If the operation fails with error code EBADF, the descriptor has not been opened and we can safely ignore it (395). This solution is quite naive but works very well unless the process uses a very large number of descriptors (which is also a rare situation).

After closing file descriptors one wants to put the sandbox into VFS jail. The entire operation is performed by *jail_vfs* function, whose code is presented above. There are two OS-provided mechanisms which allow for creation of secure jails:

- bind mounts — this operation 'binds' one directory in the file system to some other. In other words if one binds */usr* under */tmp/usr*, then all the contents of the */usr* directory will be visible under */tmp/usr*, while the original contents of */tmp/usr* will be shadowed; also any data written to */tmp/usr* will be written to */usr*. This is almost equivalent to */tmp/usr* being a hard link to */usr* (temporally as after unmounting it, it will still point to the same physical place on the block device), however it avoids some limitations of hard links (destination mount point doesn't need to be on the same partition etc.),
- VFS namespaces described previously.

However, the VFS namespaces normally do not provide complete isolation. In order to be more flexible the namespaces introduced a concept of mount propagation. Each mount inside kernel is represented as a mount object. These objects are naturally organized in a tree structure: at the top there is the mount object responsible for / directory, mount for */home* would be its child and mount for */home/marcin* would be a child of */home*. When one performs a bind mount in a namespace, for example mounting */home* at */tmp/home*, and in the other namespace someone adds a new mount at */home/jedrzej*, mount propagation allows seeing the new */home/jedrzej* under */tmp/home/jedrzej* in the other namespace. Stating it plainly: mount propagation allows the user to decide whether he wants to see the mounts which are below his mount points and belong to other namespaces in his namespace.

There is a number of flags governing whether the mount point will propagate mounts inside it to other mounts:

- MS_SHARED — all mounts are propagated,
- MS_SLAVE — mounts from points higher up in the hierarchy are propagated to the child, but child's mounts are not visible in mount points higher up,
- MS_PRIVATE — no mount propagation — this is the real namespace isolation,
- MS_UNBINDABLE — forbid bind mounting the mount point.

All the flags above can be combined with a special modifier MS_REC — it causes all the mount points below to share the same flag as the one set. For example marking a mount point */home* with flags MS_PRIVATE | MS_REC causes all the mount points on or below */home* to be marked private. The meaning and use cases of all flags except for the last one (MS_UNBINDABLE) should be easy to comprehend. The last one is used for a totally different purpose. Why would one forbid bind mounting a mount point? The usual reason is to avoid infinite recursion. Lets imagine that one mounts / under */tmp/jail*. Normally this should cause an infinite number of directory levels of the form */tmp/-jail/tmp/jail/tmp/jail/tmp/jail/tmp/jail* and so on. This could lead to DOS attacks on kernel. However if one marks the */tmp/jail* as unbindable this will be avoided — after a mount as in the example the user will see the usual contents (as if the mount has not happened) of */tmp/jail* in */tmp/jail/tmp/jail*.

After this exhaustive introduction we are able to start the analysis of *jail_vfs* function. It starts by marking the new root directory of sandbox and its parent as unbindable — this is done in order to avoid infinite recursion problem mentioned above, but it is also required by *pivot_root* function used later. In the next step one of the following two operations is performed:

- if mount type is not equal to BIND_SELECT, it mounts the entire / under the directory used as root in the sandbox (and at the same time marking / recursively private),
- if the type was specified as BIND_SELECT then the directories specified by the user are mounted below jail root directory using *bind_in_jail* function.

After these steps we mount the private */tmp* directory inside the jail root (440) — no matter what option we choose, the sandbox will not have access to global */tmp*. Next we jail the process inside the jail root directory. This is done by using *pivot_vfs* function (444). Inside this function we call a *pivot_root* system call (91). The call changes the process' / directory to the one specified as the first argument and leaves the old / mounted at the path pointed to by the second argument. It has the following requirements:

- the new root as well as its parent must be mount points and must be marked as unbindable,
- the second argument (path to the old root mount point after the call) must be a child of the new root directory. In other words if the new root is */tmp/jail* than the path to the old root mount point must inside "*/tmp/jail/...*".

Pivot root is a point of no return. After umounting the old root (line 92) the process cannot escape its VFS jail as it has no reference to the original root directory. When using normal chroot system call the process kernel structure (*task_struct*) keeps a reference to the original / and only interprets filesystem calls (like *chdir*) differently. After *pivot_root* all pointers inside *task_struct* direct to the directory specified as first argument of the call as the root of VFS.

Lastly, if the user specified the type to BIND_EXCEPT a specially prepared empty directory with no access permissions for the jailed process is mounted under directories of the user's choice (449–453).

6.4. VFS resources used by sandbox

In our solution each time the sandbox is created one needs to create a set of temporary directories for use inside the sandbox. In this section we will present the hierarchy of these directories and their use. Below is a tree of directories used by a single sandboxed process:

```
/
  /tmp
    /jail
      /cgroup                -- whole system cgroup
        /jail                -- home of all sandboxed groups
          /jail-XXXXXX       -- single sandboxed process
            /cpuacct.usage
              /tasks
              /memory.memsw.limit_in_bytes
              /device.allow
              /device.deny
      /cleanup
        /jail-XXXXXX
        /jail-XXXXXX.fifo    -- fifo for messages to supervisor
      /root                  -- used as a new '/', bind mount
                                directories beneath
        /tmp                 -- used to mount per process /tmp
```

The */tmp/jail* directory is the directory where all the activity happens — for security reasons all the sandboxing operations are performed only in its sub-directories.

For each sandboxed process we create a special directory called */tmp/jail/cleanup/jail-XXXXXX*, where the last 6 'X' characters are the identifier of the process created randomly (*mkstmp* call). The contents of this directory are seen inside the sandbox as the contents of */tmp*. In other words each sandbox has a private */tmp*.

When the untrusted process starts it switches the root of the file system (*/*) to the */tmp/jail/root*. So the process can only see the contents of this directory after switching the root. Depending on sandboxing options one of the following can happen:

- the process will see the entire normal file system (read-only mode depending on options chosen),
- the process will see the entire file system except for certain directories — they will be visible but their contents will be unavailable,
- the process will see only the directories specified in the options.

What is common to all these scenarios is the fact that all visible directories will be mounted under */tmp/jail/root*. Furthermore, the directory */tmp/jail/cleanup/jail-XXXXXX* will be mounted at */tmp/-jail/root/tmp* — this asserts that each sandbox has its own, private */tmp*.

When the library uses supervisor (this is optional, however recommended) while creating the sandbox it creates a named pipe at */tmp/jail/cleanup/jail-XXXXX.fifo*. Writing data to it is a signal to the supervisor that it should destroy the sandbox it is supervising and perform the clean-up. This is much a better mechanism than sending signals to supervisor as it allows for much more granular access control. For example signals can only be sent to some process by a process with the same uid, however we can control access to a fifo simply by file permissions. This allows us to create a supervisor running with root privileges, able to react to messages sent by normal users.

The */tmp/jail/cgroup* directory is used to mount the cgroup file system (using system call equivalent to: mount -t cgroup -o memory,devices,cpuacct dummy */tmp/jail/cgroup*). Each time we set limits for a process involving cgroups we need to create a new cgroup for the process. This is done by creating a directory */tmp/jail/cgroup/jail/jail-XXXXXX* (the documentation states that mkdir call inside a cgroup mounted files system is the only way to create a new group of processes). Again thanks to putting all the jailed processes inside a single "cgroup/jail" we avoid breaking other groups created by the administrator. Also as all groups in directory */tmp/cgroup/jail* were created as part of sandboxing it is usually safe for the system to destroy them (and kill processes within) — crucial processes are usually not sandboxed there. The exact operation of cgroup interfaces we use is as follows:

- in order to limit the amount of memory and swap for the sandbox we write the value of the limit into memory.memsw.limit_in_bytes file. The interface supports 'k' suffix to the number — the value is interpreted as kilobytes allowed,
- to add a process to a cgroup we write its pid to tasks file inside the cgroup,
- to provide/forbid access to devices we write the device and access specification (as described in the section covering the API) to either device.allow or device.deny file,
- to limit the CPU usage by the sandbox we need to use supervisor. The file cpuacct.usage contains the total CPU time used by the sandbox (in nanoseconds). The supervisor periodically (every 1s) reads its contents. If the sandboxed processes exceed their CPU time limit then supervisor kills all the processes in the sandbox and does the cleaning.

6.5. Supervisor

Apart form the library we decided to provide an additional binary called supervisor. It is an optional part of our sandboxing solution. Its purpose is to supervise the sandboxed process and perform sandbox tear-down (release the resources used by sandbox) when the process inside dies. The tool has to be executed with root privileges, so we have put a lot of effort into its simplistic design. The supervisor is an optional module, but when it is not used, the user has to release some of the resources used by sandboxing mechanism manually.

To be precise the functions provided by the supervisor are as follows:

- observe the CPU usage by the sandboxed process and kill it if it exceeds the limit,
- listen on a named pipe for any data to read; incoming data is interpreted as a command to kill the sandboxed process,
- when the process exits or is killed rename the temporary directories used for file system jailing, and remove cgroup to which the process belonged.

The supervisor is executed with 7 arguments in the following order:

- path of the directory in which the child's root directory resides,
- child's pid,
- file descriptor number of an opened eventfd channel,
- file descriptor of an opened named pipe,

- user id of the process inside the jail,
- group id of the process inside the jail,
- CPU time limit for the sandboxed process (this is optional).

Crucial pieces of code are analysed below.

```
135 CHILD_PID = atoi(argv[2]);
136 child_fd = atoi(argv[3]);
137 FIFO_FD = atoi(argv[4]);
138 UID = (uid_t)atoi(argv[5]);
139 GID = (gid_t)atoi(argv[6]);
140
141 if (argc > 7)
142  CPU_limit = atoll(argv[7]);
143
144 if (child_fd < 1 || FIFO_FD < 1)
145  return 1;
146
147 BAD_RETURN(sigaction(SIGCHLD, &act, NULL), -1, -1);
148 BAD_RETURN(write_event(child_fd, 1), -1, -1);
149 close(child_fd);
150
151 BAD_RETURN(PATHS = create_jail_paths(), NULL, -1);
152 BAD_RETURN(init_jail_paths(PATHS, path), -1, -1);
153 if (CPU_limit > 0)
154  BAD_RETURN(mount_cgroupfs(), -1, -1);
155
156 BAD_RETURN(event_loop(PATHS, CHILD_PID, FIFO_FD,
157                       CPU_limit, UID, GID),
158            -1, -1);
```

Lines 135–146 basically read the command line arguments. In line 147 we set up a handler which will be fired when the sandboxed process dies. It is required so that the supervisor is ready to perform the cleanup when the sandboxed process exits. When the handler is set the supervisor writes a message to an eventfd channel to inform the sandboxed process that it has set up proper handler and the child can continue. Later, the supervisor prepares the cgroup interface for CPU accounting if the CPU time limits are set (lines 153–154). At last the supervisor enters a busy wait loop in which it waits for messages (the *event_loop* function in line 156).The crucial piece of *event_loop* function is presented below.

```
92  while ((death == 0) && (!time_kill) && (kill_msg == EV_TIME)) {
93   death = wait_nonblock(child_pid);
94
95   if (death < 1)
96     time_kill = check_CPU(paths, CPU_limit);
97
98   if ((death < 1) && (time_kill == 0))
99     kill_msg = wait_fifo(fifo_fd, S_TIME);
100 }
```

The first action of the supervisor is checking that the child is still alive (line 93). Later on it uses cgroup interface to check how much CPU time the child had used so far (*check_cpu* function in line 96). If the child exceeds the limit it exits the loop. Lastly, the supervisor waits (using select system call) for input on the named pipe. The wait itself happens inside *wait_fifo* function (line 99) and is limited to 1 second — thanks to that the CPU usage can be checked every second.

The last important piece of functionality of supervisor is its ability to clean-up after the sandbox has been thorn down. It is implemented in the *cleanup* function:

```
23 int cleanup(struct jail_paths *paths, int fifo_fd, uid_t uid,
              gid_t gid) {
24   close(fifo_fd);
25
26   BAD_RETURN(chdir("/"), -1, -1);
27   BAD_RETURN(cleanup_tmp_jail_dirs(paths), -1, -1);
28   //this actually can fail - it is harmless,
     so we don't even bother to
29   //check if the cgroup for sandbox was ever created
30   BAD_RETURN(cleanup_cgroupfs(paths, uid, gid), -1, -1);
31   free(paths);
32   paths = NULL;
33   close_fds(NULL);
34
35   return 0;
36 }
```

Firstly the unneeded file descriptor is closed and the working directory is changed to root (lines 24–26). This is to make sure that the virtual file system resources are freed, so that we will not get the 'Resource is use' error message when we try to rename or remove directories. In line 27 the temporary directory used as */tmp* by the sandboxed process is renamed to a special name, so that it will be easy to identify and delete. Normally one could remove the directory and its content. However, such operation can be potentially very dangerous in case of an error. We decided to perform a simple rename, because such operation is reversible on one hand and the removal of unused directories with a specific path pattern is easy at any point on the other. That gives us both security and ease of cleaning up temporary files by the administrator whenever he/she deems fit. A much nicer solution for this would be to use tmpfs which is stored in RAM and is automatically cleaned. However, in Linux there are still no quotas on tmpfs, so using it would allow for a denial of service attack (by exhausting available memory).

Lastly, the process closes all the file descriptors it has opened (line 33).

6.6. SUID Wrapper

The SUID binary was developed for the purpose of providing an easy to use tool for testing the library, and demonstrating its abilities from the command line. As such the design was marked by the goal of simplicity of implementation and usage. Developing full-featured tool in C would be prohibitively error prone and time consuming. We decided to build a very short C binary which executes a Python script (the path is hard coded for security reasons). The script is responsible for parsing all the command line arguments and calling library to perform the required actions. We make only very conservative assumptions about Python version (≥ 2.4) — thanks to that the tool is portable to older systems.

Using a high-level scripting language to build the tool has two other fundamental advantages:

- thanks to the expressive power of the language it was very easy to build and intuitive and very expressive syntax for command line parameters,
- the tool demonstrates how the library can be used from a high level language using only simple C bindings.

For the purpose of calling C library functions form Python the jail.py script uses the *ctype* library. It belongs to a standard Python distribution. The library is particularly well suited for the task as it can handle both string and integers conversions between Python and C and it has a support for pointers. C-language types are handled by the following Python classes:

- *c_char_p*,
- *c_int*,
- *c_void_p*.

In order to make use of *libjail.so* one has to load it into Python process environment. *Ctype* achieves that by providing a library loader object *cdll*. The object is supplied with *LoadLibrary* method which takes a file name as its only argument. The method loads the dynamic library from file specified by the argument and returns a dynamic library object. Functions provided by the loaded library are accessible as attributes of the dynamic library object. This is easily illustrated by the following excerpt of code:

```
#load the library by specifying the name of the file
lib = cdll.LoadLibrary("libc.so.6")
#use the printf functions from the library
#the code below will print "Hello world" on standard output
#just as if it was written in C
lib.printf("Hello world")
```

If the arguments supplied to C function calls are of basic types (strings, integers) they are automatically converted to appropriate ctypes (*c_char_p/c_int*). Return values of pointer types are visible as *c_void_p* arguments. The library API was purposely designed to use only pointers, integers and strings. All the type conversion happens transparently and requires no further coding.

Command line arguments are handled by *optparse* library. It was chosen because it has belonged to the Python standard distribution for a long time. The command line parsing code is in most cases very standard. The only noticeable feature are the security checks performed by the script. The checks are omitted if the real user id is equal to 0 (which means that the user has root privileges).

Below there is a list of restrictions that the jail.py script asserts on the user in order to avoid simple cases of misuse:

- do not allow changing user id or subscribing to a group to which the user does not belong,
- do not allow changing the rlimit limitations to values higher than set for the parent process.

6.7. Porting suEXEC

We have chosen to demonstrate how existing software can be improved by making use of our sandboxing solution on the example of suEXEC. suEXEC is an Apache module that provides ability to run CGI and SSI programs under user's uids (instead of the uid of Apache itself) — more details can be found in [14]. It is designed to reduce security risks connected with allowing users to develop and run private web programs considerably (this is an exemplification of a more generic idea called "confused deputy problem", its description can be found in [10]).

6.7.1. suEXEC architecture

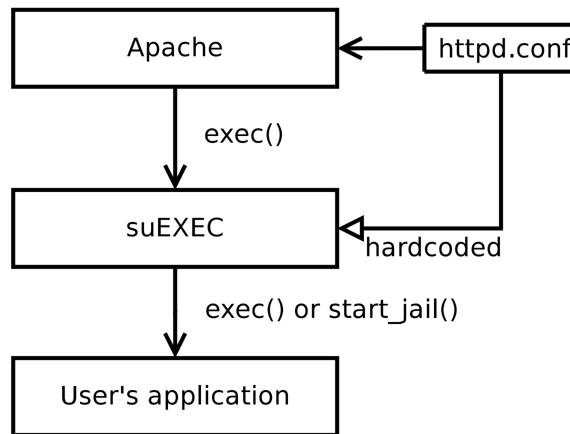

Figure 6.1: process of calling user's application with suEXEC

When started, Apache looks for *suexec* binary in its bin directory. If found, suEXEC will be used to run user's applications from now on.

suEXEC is designed as a single *.c* file that compiles to a single SUID binary. The path to user's docroot directories (root of the user's webpage) is hardcoded in the binary and cannot be changed by modifying only the httpd.conf file. Apache invokes suEXEC with three parameters:

1. [~]user — specifies under which user's privileges the code should be executed (suEXEC looks for a CGI/SSI file in user's docroot directory when '~' proceeds the user name otherwise — in the main docroot),

2. group — specifies under which group's privileges the code should be executed,

3. command — relative path to the file (from the docroot).

suEXEC performs the following steps:

1. Cleans the environment.

2. Checks validity of the arguments (existence of corresponding entries in */etc/passwd*, existence of docroot directory, occurrences of .. in command argument, etc.).

3. Drops privileges (*setuid*, *setgid*).

4. Checks validity of the arguments (changes directory to docroot and back, finds the command file, checks its permissions and ownership, etc.).

5. Closes all file descriptors.

6. Executes the code (*exec*).

suEXEC architecture is demonstrated on figure (6.1).

6.7.2. Sandboxing in suEXEC

The main goal is to substitute execution (*exec*) with sandboxing (*start_jail*). It is not a straightforward task, as in step (6) suEXEC has already dropped privileges and works with user's privileges.

Performing all validity checks in superuser mode can be insecure even though it might seem totally harmless. Consider the following situation: an exotic filesystem supported by fuse is mounted to */mnt/exotic*. Chdir operation on this system is defined in such a way that it automatically creates a new directory (if one does not already exist). Now suppose there is a security check that uses chdir to get into some subdirectory of user's home folder. The user might have created a symbolic link from that subdirectory to */mnt/exotic*. In consequence the code would change the filesystem, to which the user should not have any access.
To preserve all the properties of the current code it is necessary to implement the scheme depicted on figure (6.2).

6.7.3. Patch

In our patch we implement the aforementioned scheme depicted on figure (6.2) to ensure that all security checks are made with minimal privileges. For the communication between root and user's processes we use pipes.

The original use of *exec* function was substituted with the following code:

```
struct jail_param *jail = create_jail_param();

set_exec_path_param(jail, cmdpath);
set_pid_namespace_param(jail, 1);
add_exec_arg_param(jail, argv[3]);
set_mount_type_param(jail, 1);
set_id_param(jail, uid, gid);
set_fd_type_param(jail, 0);
set_env_type_param(jail, 0);

/*
 * disallow access to all devices except from:
 * /dev/null, /dev/zero, /dev/random,  /dev/urandom, /dev/tty0
 * /dev/tty, /dev/console and /dev/ptmx
 */

add_cgroup_device(jail, 0, "a");
add_cgroup_device(jail, 1, "c 1:3 rwm");
add_cgroup_device(jail, 1, "c 1:5 rwm");
add_cgroup_device(jail, 1, "c 1:8 rwm");
add_cgroup_device(jail, 1, "c 1:9 rwm");
add_cgroup_device(jail, 1, "c 4:0 rwm");
add_cgroup_device(jail, 1, "c 5:0 rwm");
add_cgroup_device(jail, 1, "c 5:1 rwm");
add_cgroup_device(jail, 1, "c 5:2 rwm");

/*
 * set cpu/mem restrictions to 60 seconds of CPU and 200MB of memory
 */
```

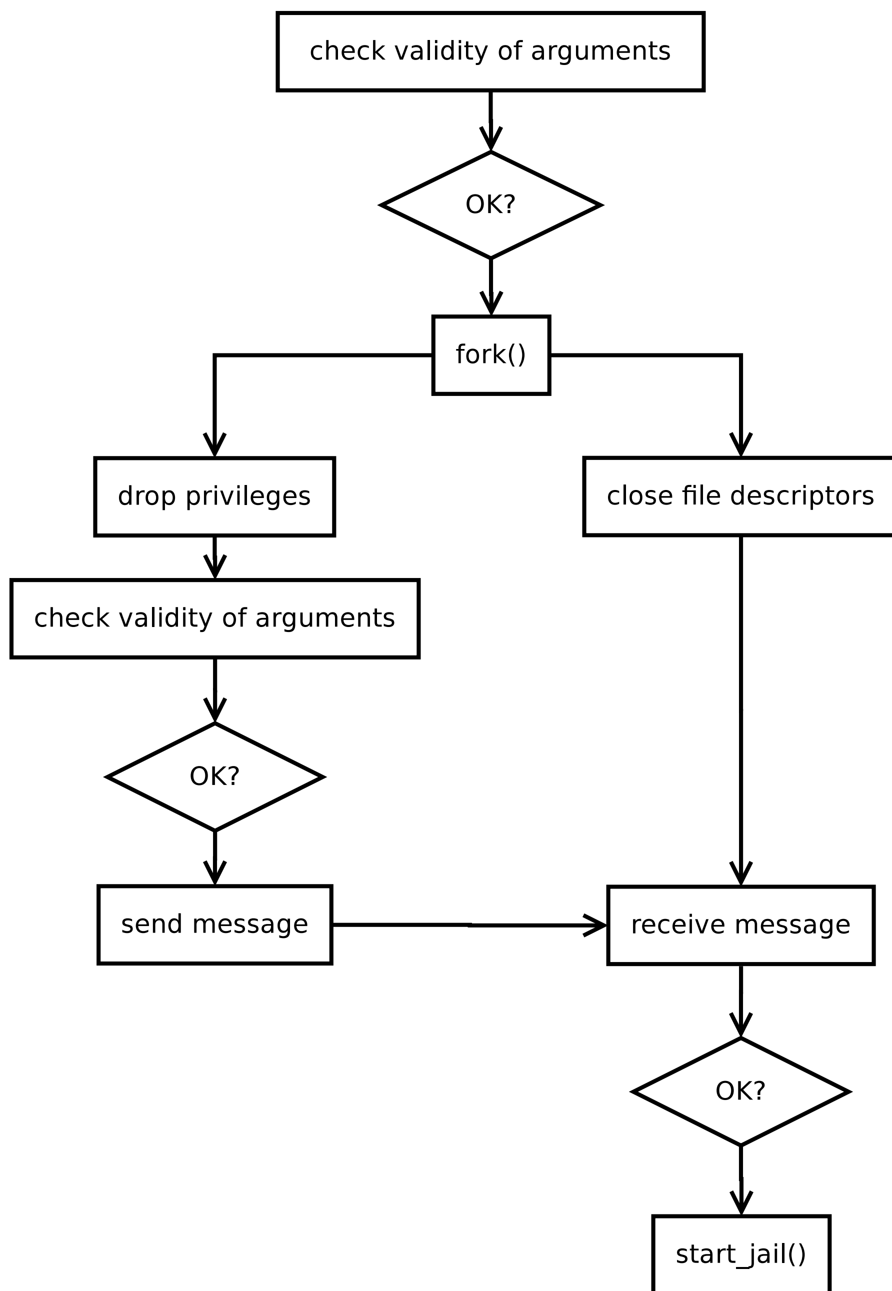

Figure 6.2: Preserving superuser capabilities while performing operations with user’s privileges.

```
set_cgroup_cpu(jail, 60000);
set_cgroup_mem(jail, 1024*200);

/* execute the code using supervisor */

pid_t jailed = start_jail(jail, 1);

/* wait for CGI/SSI to finish */

if(jailed>0) {
    wait(NULL);
    exit(0);
}
```

First a *jail_param* object is created. Afterwards, in the first section the path to the executable is set, PID namespaces option is turned on, arguments are passed, UID and GID are set, filesystem is mounted RO and finally descriptors and environment stay unchanged. suEXEC itself cares about descriptors and cleans the environment. The original code is well tested so we decided not to change it.

In the second section we deny access to all devices excluding: */dev/null, /dev/zero, /dev/random, /dev/urandom, /dev/tty0, /dev/tty, /dev/console,* and */dev/ptmx.* We considered all other devices to be unneccessary for the web scripts.

Third section sets CPU and memory limits to 60 seconds and 200 MB respectively.

After *jail_param* is prepared *start_jail* is called with the supervisor flag on. In the end the child is waited for if *start_jail* has succeeded. Otherwise an error is raised.

6.8. Summary

After reading the chapter, the reader should have some basic understanding of the techniques used by the authors while creating the sandbox. Furthermore, the chapter covered the public APIs and purposes of each of the components of the solution. The next one will concentrate on tests performed using the sandboxing tools and the analysis of their results.

Chapter 7

Tests

In this chapter we will cover how our sandboxing solution has been tested. The tests are divided into two separate groups: correctness and performance. The task of the first group is to ensure that the mechanisms are implemented properly, and the goals concerning security have been achieved. The second group, performance tests, has been run to ensure that the overhead of our solution is not larger than 10%.

7.1. Testing platform

All the tests were performed on Intel Core 2 Duo (T7300) machine with 2GB of RAM memory and no swap partition. The system used was Linux Slackware 13.37 x86_64 running on kernel 3.0.9. It features libc version 2.13, gcc 4.5.2 and Apache 2.2.21.

To perform the tests a set of simple programs exploiting specific system resources was created. They were used in both correctness and performance tests.

7.2. Correctness tests

Extensive correctness tests of the C library were performed using the SUID wrapper. Two additional tests covered functionality of the suEXEC patch.

7.2.1. Memory limits

We have created two types of memory limits to cover the following two cases:

1. a single process exceeds the memory limit,
2. group of processes exceeds the memory limit, but none of the processes does it by itself.

7.2.1.1. Single process test

The first and the most basic memory test (*allocate.c*) was done by allocating a given amount of memory, filling it with data and computing a given function of that data. If the correct answer is returned it can be assumed that the process got sufficient amount of memory.

The mechanism of cgroup kills the group of processes when the limit is exceeded. Rlimit, on the other hand, prevents from allocating the memory by returning NULL every time the limit would be exceeded.

The code of the test has no checks for the return value of malloc:

```
13 mem = (int*) malloc( mb_to_allocate * (1024*256*sizeof(int)) );
14 for(it=0; it<1024*256*mb_to_allocate; it+=256) {
15     mem[it]=(it*it+3)%(mb_to_allocate*1024*256);
16 }
```

It is expected that exceeding rlimit will result in the kernel killing the process with SIGSEGV. In case of cgroup OOM killer (a kernel tool used for freeing memory for the system) will take care of it.

As expected running

```
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
    --limit_as 10000000 -- allocate 11
```

results in SIGSEGV

```
kernel: allocate[3188]: segfault at 0 ip 0000000000400689
    sp 00007fffc1f17340 error 6 in allocate[400000+1000]
```

And analogously running

```
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
    --mem_max 10000 -- allocate 11
```

results in OOM killer action

```
kernel: Task in /jail/jail-v5KmaA killed as a result of limit
kernel: memory: usage 9892kB, limit 10000kB, failcnt 16
kernel: memory+swap: usage 0kB, limit 9007199254740991kB, failcnt 0
kernel: [pid] uid  tgid total_vm  rss cpu OOM_adj OOM_score_adj name
kernel: [3210]  1000  3210  3573  2519  1  0  0 allocate
```

Allocate executed with argument $\leqslant 10$ (not exceeding rlimit) finishes its task successfully with 10MB memory limit.

7.2.1.2. Multiple processes test

The second test is very similar: the process forks and performs two independent tests of the first kind. The interesting case is when $Alloc < Lim < 2 \cdot Alloc$. It is expected that in this case the process will be killed when using cgroup limit, but will continue and return correct result when using rlimit.

In fact running

```
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
    --limit_as 10000000 -- allocate2 8 8
```

results in obtaining the answer

```
[INFO] to_allocate= 8+8
[INFO] meaningless: 3
[INFO] meaningless: 3
DONE
```

while using cgroups

```
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
    --mem_max 10000 -- allocate2 8 8
```

results in OOM killer action

```
kernel: Task in /jail/jail-MGINeS killed as a result of limit
kernel: memory: usage 10000kB, limit 10000kB, failcnt 35
kernel: memory+swap: usage 0kB, limit 9007199254740991kB, failcnt 0
kernel: [pid] uid tgid total_vm rss cpu OOM_adj OOM_score_adj name
kernel: [3686] 1000 3686 3061 1324 0 0 0 allocate2
kernel: [3687] 1000 3687 3061 1266 1 0 0 allocate2
```

7.2.1.3. Results

Memory test results are shown in the following table:

Type of the test	Memory allocated	Type of limit	Limit	Action	Result
single process	10	cgroup	100 MB	answered	PASSED
single process	10	cgroup	10 MB	answered	PASSED
single process	10	cgroup	9 MB	OOM killer	PASSED
two processes	10+10	cgroup	20 MB	answered	PASSED
two processes	10+10	cgroup	15 MB	OOM killer	PASSED
single process	10	rlimit	100 MB	answered	PASSED
single process	10	rlimit	10 MB	SIGSEGV	PASSED
two processes	10+10	rlimit	20 MB	answered	PASSED
two processes	10+10	rlimit	15 MB	answered	PASSED
two processes	10+10	rlimit	10 MB	SIGSEGV	PASSED

It proves that the memory limit mechanism works as in the specification. Sandboxed programs are terminated if any process exceeds its limit (in case of rlimit) and if the whole group exceeds it (in case of cgroups).

7.2.2. CPU usage limits

7.2.2.1. Tests description

In analogy to memory usage, there are two basic CPU usage tests: one with a single process occupying the CPU for a given amount of time and one which forks before occupying the CPU in each branch.

In each case there are some computations performed by the programs in an infinite loop. Once in every 10000 iterations special file: /proc/*pid*/stat is checked for CPU usage and when the given CPU time exceeds the limit the program breaks out of the loop.

We have measured the accuracy of cgroup+supervisor mechanism. More precisely: how much CPU time can a program get when the limit is set to x. The difference between these two times depends on both the limit and the number of threads.

The same test can be performed for the rlimit mechanism.

7.2.2.2. Results

Accuracy of the cgroup+supervisor mechanism is about 2s in the worst cases. The same result can be obtained from the theoretical reasoning. In the worst case the supervisor checks cgroup accounting data 1s after the limit has exceeded. During this second n seconds of CPU time can be used by the sandboxed program, where n is the number of cores.

When the limit is larger than 50s we can assume that statistically the accuracy will be close to 0.5s. The dependency of the accuracy upon the CPU limit and the number of threads is depicted in plots (7.1) and (7.2) respectively.

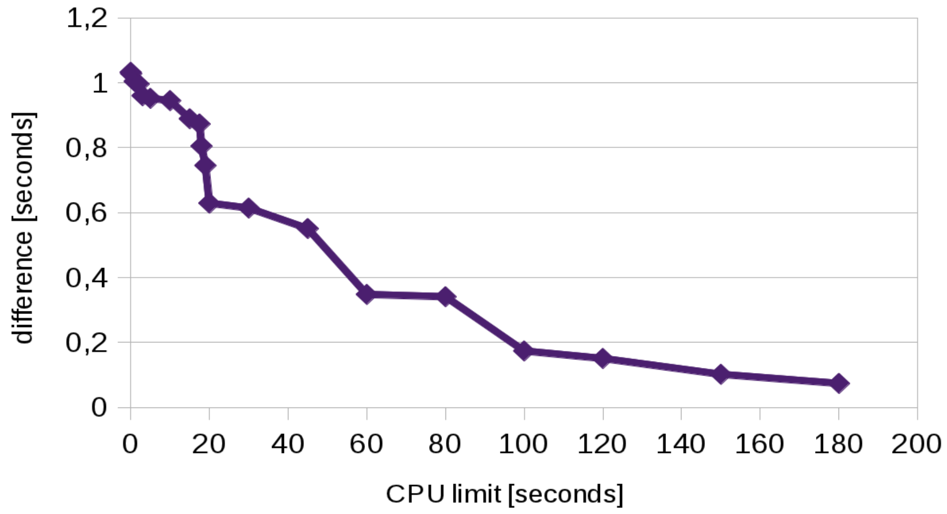

Figure 7.1: Cgroup+supervisor CPU limit accuracy — dependence upon the CPU limit in seconds

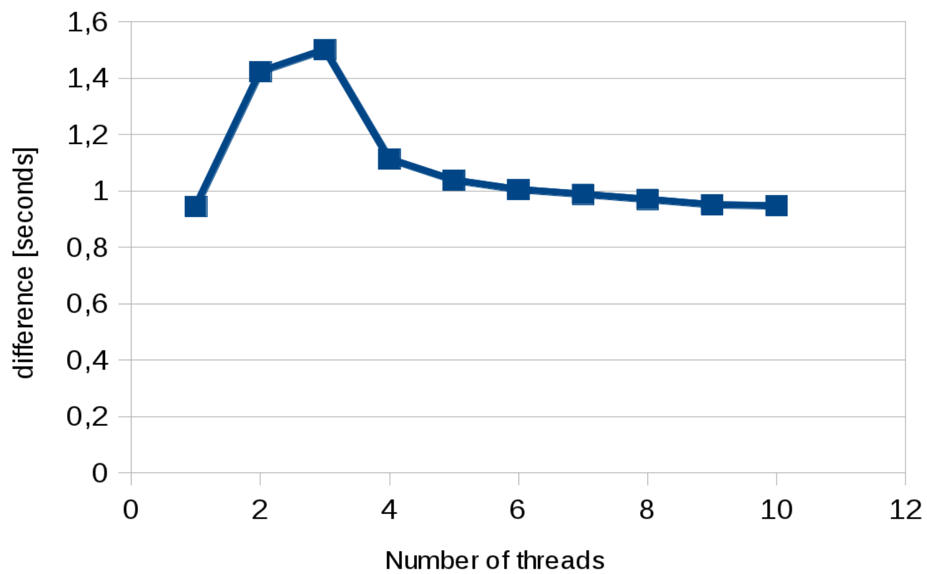

Figure 7.2: Cgroup+supervisor CPU limit accuracy — dependence upon the number of threads

The rlimit mechanism is far more accurate. It can be assumed that for any given limit the difference between the rlimit and the the real CPU time the process gets is not larger than 0.1s. However, rlimit allows integer values of seconds and works only for one thread.

7.2.3. Device access

For device access tests we have written a simple tool called *readdev.c*. It tries to perform the following operations:

1. open the device for reading,
2. read from the device,
3. seek in the device,
4. close the device,
5. open the device for writing,
6. seek in the device,
7. write to the device.

In case of failure the program prints an error message and proceeds to the next operation that may succeed. For example, after failing to open the device for reading the program moves to step (5), which is opening the device for writing.

7.2.3.1. Results

The tests were performed on */dev/zero* device, which is by default readable and writable by any user. On the kernel's side all the security checks are included in the *open* call. It is then easy to predict, that denying read- and write- access in our sandboxing mechanism will cause *readdev* to fail on *open(...,"r")* and *open(...,"w")* respectively.

```
$ readdev /dev/zero
DONE
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
   --device_deny="a 1:3 r" -- readdev /dev/zero
[INFO] could not open device for reading
[INFO] could not open device for writing
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
   --device_deny="a 1:3 w" -- readdev /dev/zero
[INFO] could not open device for writing
$ jail --id 1000:100 --vfs_all R --env_select PATH,TERM \
   --device_deny="a 1:3 rw" -- readdev /dev/zero
[INFO] could not open device for reading
[INFO] could not open device for writing
```

7.2.4. VFS restrictions

Tests concerning VFS restrictions can be performed using *readdev* tool in the following way:

1. To test whether each jailed process is given its own */tmp* directory we simply executed *readdev* on a file from */tmp* twice. During the first execution the file should be created (no error on writing), but the second instance should not be able to access it (error on reading).
2. Read-only access can be tested by simply running *readdev* on a file that is accessible for writing (according to the permissions). A failure on opening the file for writing is expected.

These two tests can be performed to test *–vfs_select* and *–vfs_except*.

7.2.4.1. Results

Result of 'empty' mount test is as follows:

```
$ ./jail --id 1000:100 --vfs_all=R -- readdev /tmp/x
[INFO] could not open device for reading

$ # at this point we should have the file /tmp/x created

$ ./jail --id 1000:100 --vfs_all=R -- readdev /tmp/x
[INFO] could not open device for reading
```

As expected the file created in one instance of sandbox is not visible in another.

Result of read-only mount test is as follows:

```
$ ./jail --id 1000:100 --vfs_all=R -- readdev /var/tmp/x
[INFO] could not open device for reading
[INFO] could not open device for writing
```

As expected creating a file in */var/tmp* has failed, even though the directory has read-write-execute permissions for owner, group and others.

7.3. Performance tests

Cgroup mechanism, supervisor and vfs namespaces influence performance to a certain degree. In this chapter we find the cost of using our sandboxing solution.

7.3.1. Cost of execution

First we evaluate the cost of executing an empty program

```
int main() {
    return 0;
}
```

using jail SUID wrapper with only the default arguments

```
$ time jail --id 1000:100 --vfs_all=R --env_select PATH,TERM -- empty
real    0m1.098s
user    0m0.060s
sys     0m0.008s
```

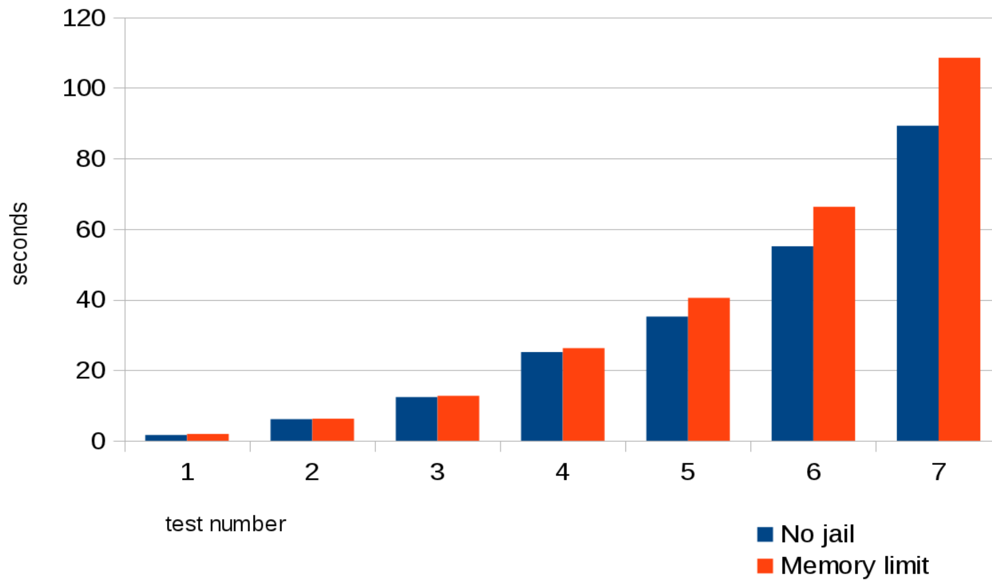

Figure 7.3: Memory limit overhead — dependence upon the amount of memory to allocate

Taking the mean value of 30 iterations we obtain following values:

- real: 1.085s
- user: 0.044s
- sys: 0.008s

which can be compared to results of an analogous test without sandboxing:

- real: 0.001s
- user: 0.000s
- sys: 0.000s

Initialization of the sandbox has therefore a significant cost.

7.3.2. Cost of cgroup memory limiting mechanism

We have tested performance of the cgroup memory limiting mechanism by executing computations on seven data sets. In each data set a given amount of memory is allocated and deallocated 100 times. For test number n it is $\frac{1}{n} \cdot 500$MB. Each computation was run three times in two different settings:

- with sandboxing and cgroup memory limit,
- without sandboxing.

The mean values of these three measurements are depicted in the plot (7.3). The overhead increases with the amount of memory that is being allocated, but can be considered negligible under 100MB.

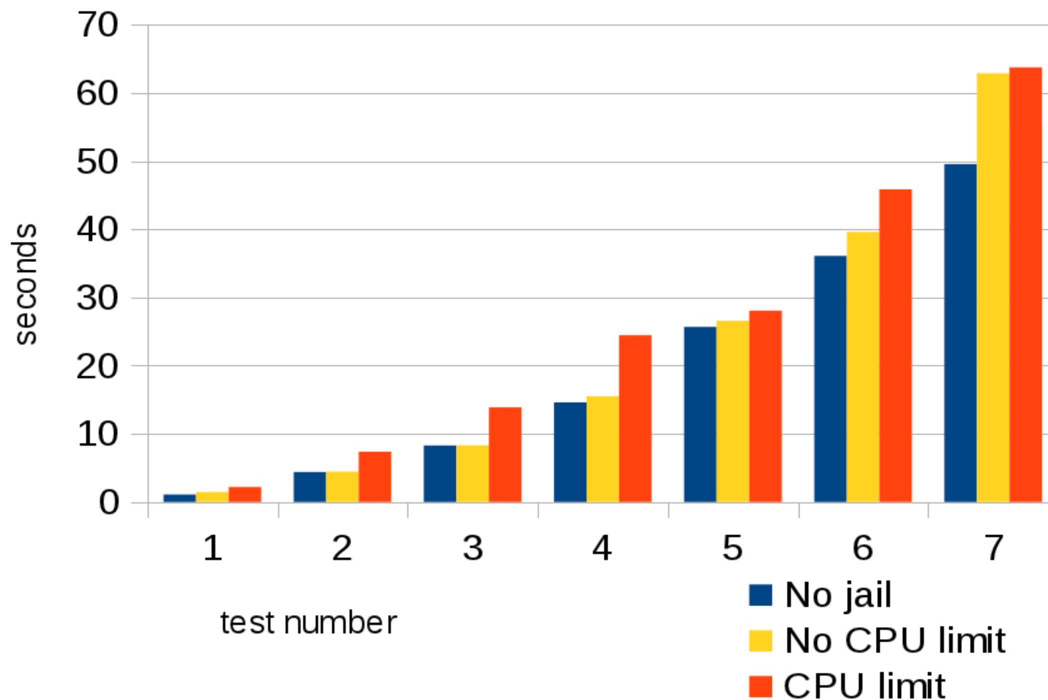

Figure 7.4: CPU limit overhead — dependence upon CPU usage

7.3.3. Cost of supervisor and cgroup CPU accounting

We have tested performance of the supervisor and cgroup CPU accounting by executing computations on seven data sets. For test number n the value of π (ratio of cricle circumference to its diameter) was evaluated to $500(1+n)$ digits of precision. Each computation was run three times with three different settings:

- with sandboxing and CPU usage limit,
- with sandboxing, but without CPU usage limit,
- without sandboxing.

The mean values of these three measurements are depicted in the plot (7.4). The overhead depends highly on the specific data. In each case sandboxing with CPU limits is slightly slower than sandboxing without CPU limits, which on the other hand is slower than execution without sandboxing. For low CPU usage sandboxing itself influences the performance much less than CPU accounting and the supervisor. For high CPU usage sandboxing seems to cause majority of the overhead.

7.3.4. Cost of VFS namespaces

We have tested performance of VFS namespaces on three programs massively using the filesystem. Each test was run seven times with two different settings:

- with sandboxing (and automatically with VFS namespaces),
- without sandboxing.

	with sandboxing			without sandboxing		
Test number	real	user	sys	real	user	sys
1	4.713	0.160	0.364	2.625	0.120	0.340
2	6.089	0.336	0.772	5.524	0.264	0.756
3	10.084	0.552	1.348	9.941	0.536	1.288

Table 7.2: VFS namespaces overhead

The testing involves creating and removing k^2 files, where $k = 50, 75, 100$ for tests number $1, 2, 3$ respectively. Before executing each test the cache was emptied by calling 'echo 3 > /proc/sys/vm/-drop_caches'. The mean values of those seven measurements can be found in the table (7.2). As the results show, the overhead decreases with the complexity of the test.

7.4. suEXEC tests

To test suEXEC we have written a script *index.cgi* that covers all functionalities of our patch by using all the forbidden resources. It tests the following features:

1. pid namespaces - simply by calling '*ps aux*',
2. device access - by writing to '*/dev/full',*
3. cgroup memory limit - by invoking '*tests/allocate 220*',
4. cgroup CPU limit - by invoking '*tests/cpu2 61 61*',

An example of correct output:

```
User: uid=1000(olorin) gid=100(users) groups=100(users)

ps aux:
USER PID %CPU %MEM   VSZ   RSS TTY     STAT START   TIME COMMAND
root   1 0.0 0.0 8360  340 ? S 16:29 0:00 supervisor /tmp/jail/ ...
user   2 0.0 0.0 17600 1416 ? S 16:29 0:00 ~/.homepage/index.cgi ...
user   5 0.0 0.0 15136 1096 ? R 16:29 0:00 ps aux

write to /dev/full: /home/crypted/users/olorin/.homepage/index.cgi:
  /dev/full: Operation not permitted
```

The first line implies that the script was executed under the correct user privileges. Second section proves that the process had been closed in a pid namespace. Third section shows how device usage is restricted.

As the script invokes *allocate* and *cpu2* it can be deduced (from the lack of output) that the subprocesses were killed by OOM-killer and supervisor respectively, which proves the corresponding limits were set properly.

7.5. Summary

Our sandboxing solution has passed all the correctness tests. In terms of performance the overhead remains within 10% for programs which heavily use CPU, memory and VFS. On the other hand, the

time needed for initialization of the sandbox is significant and for that reason this tool is not suitable for frequent *start_jail* calls.

Chapter 8

Summary

8.1. Goals achieved

We have successfully created and tested the following components: C library (providing tools for sandboxing), SUID wrapper (commandline application for running programs in sandboxes), supervisor (process used for monitoring sandboxed programs), suEXEC patch (improving suEXEC so that it is able to run a sandbox for each CGI/SSI script).

8.1.1. C library

C library is the substantial part of our project. The main goal of it was to make our sandboxing solution applicable in all programs written in C and higher level languages. This goal was achieved as can be demonstrated by examples of suEXEC patch (written in C) and SUID wrapper (written in Python). The library implements various features such as:

1. programmer-friendly environment, argument list, file descriptors, uid and gid list handling (manual modification of these takes many lines of code and demands special care),
2. pid namespaces, rlimit settings, cgroup memory (most of these can be set in one line),
3. cgroup CPU limits (a feature that is not implemented in kernel),
4. VFS manipulation (flexible enough way of binding directories RO/RW/empty).

Moreover the library takes care of 'little details' that seem reasonable for almost all cases of sandboxing and can be done automatically. Examples of such are: dropping capability bounding set, locking secure bits, mounting empty /tmp directory.

From the programmer's point of view this component (include file *api.h* and the shared library *libjail.so*) is such a simplification of secure sandbox execution that it can be applied instead of almost every *exec* call.

8.1.2. SUID wrapper

SUID wrapper has been implemented in Python (it demonstrates how the library can be used in high level languages). Its main purpose is to provide a sufficiently flexible method of creating sandboxes from the commandline. It is very powerful as it covers the majority the library's features. It also performs a large number of security checks to avoid being misused by an inexperienced user.

8.1.3. Supervisor

The supervisor process has been created as a workaround to the problem of lacking cgroup CPU limits. This component is used for monitoring CPU usage inside a given cgroup and react adequately. This is currently the only solution that allows limiting CPU time for a group of processes. Due to the complexity of this system the precision is not perfect (in the sense that a group can exceed the limit and be killed a few hundreds of milliseconds later). Nevertheless, more precise limitations are very rarely necessary.

8.1.4. suEXEC patch

suEXEC (Apache component responsible for executing CGI/SSI scripts under user privileges) has been modified in order to use our sandboxing library. The main goal of suEXEC (which is to isolate each user's webpage) was never fully achieved. suEXEC restricts the interference between scripts, but it uses tools which are too primitive to provide sufficient isolation. Our patch reduces the interference on the levels of interprocess communication (by pid namespaces), filesystem (by separating */tmp* directories), server resources (by limiting CPU/memory usage), and external devices (by disallowing device usage).

8.1.5. Performance

Performance of our sandboxing solution is satisfactory. Thorough testing proved that the overhead does not raise above 10% of real time of execution for programs that make heavy use of CPU, memory and VFS. On the other hand the time of initialization is not negligible. Therefore our solution is not suitable for use-cases where very frequent execution of external code is necessary. In most situation, however, it is possible to reduce the number of executions.

8.2. Further development

Sandboxing is a very wide concept and the need for tools supporting it grows with the development of distributed technologies. Therefore we only present some of the numerous ideas for extensions, which we consider the most important.

1. Provide a solution that obsoletes SUID binaries. For now creating new cgroups, namespaces, and VFS jails requires full root privileges. Allowing the user for these operations requires either changing a number of core kernel functions or using SUID binaries. Avoiding the latter makes it necessary to give ordinary users the right to bind-mount directories, which seems not only difficult, but also unlikely to be merged into the mainline kernel, as it would require changes in core VFS layer. Such changes would have been supported by important use cases, as kernel developers are reluctant to bug-prone code in that module. On the other hand allowing users to create namespaces seems a very realistic possibility, some work on pid namespaces for users has already been done. As far as cgroups are concerned, allowing users to create them seems reasonable as well as beneficial. We suppose that building a sandbox solution with the same strength as ours which calls SUID binaries only for setting VFS jails is possible.

2. Create a cgroup which kills process groups exceeding their CPU time limit. Currently for this task we use the supervisor. However, this seems slightly awkward as we have to create another process only to observe CPU usage. This process must wake up every now and then to check the counters, which influences the performance negatively. In case of mobile users (smart phones/laptops) such wakeups draw precious energy and should be avoided. Creating a

kernel mechanism — just like the memory usage limit — should be a straightforward task as the counters have already been implemented. It is only the handler for the event of exceeding the limit that is left to be written.

3. Allow for recursive read-only bind mounts. Current kernel implements read-only bind mounts, but the read-only flag is not recursive. The easiest way of explaining the problem is by the following example. Assume that */home/marcin* is mounted via NFS. When the user requests to recursively bind mount */home* under */tmp/home* with read-only property, the mount will succeed. The content of */tmp/home* directory will be read-only, except for subdirectories which are mount points. That means the directory */tmp/home/marcin* will be mounted read-write. As a result the read-only flag in our sandbox has to be used carefully used as it is not always enforced and its usage depends on the system configuration. The sandbox needs different settings when */etc* is remotely mounted and different when it is simply a directory under /. Changing kernel implementation of recursive mount seems possible however, as it touches crucial functionality, it would require very thorough testing.

4. Provide an interface for configuring the library. Instead of using hard-coded paths to some filesystem directories, these settings could be defined in */etc* and read during sandbox execution. This would allow various Linux distributions to configure the library according to their demands. Also the configuration could enforce some other rules, such as defining a list of users/groups that are allowed to execute functions from the library, list of devices which should always be disabled, etc.

5. Port SSH server to use our library. It would to provide optional sandboxing for the shell started by SSH server. Configuration file */etc/ssh/sshd_config* would specify per user sandbox limitations. This would strengthen the security of systems with multiple shell accounts. Such systems are commonly used as git servers, as the design of git itself assumes usage of SSH for repository access. Even if the users are trusted, the probability of attack increases with their number. Consequences of a potential security breach would be more limited because of the isolation provided by the sandbox. Especially pid namespaces and limited access to devices would decrease the number of kernel/userspace interfaces that are exposed to the attack. Moreover, a specific usage pattern (as in case of git) may allow such a configuration that only the desired and very limited functionality is available in the sandbox.

6. Add network namespaces support to the library. In our current implementation we decided to omit the network namespaces. Using them effectively requires a lot of code and much of it would have to be executed with root privileges. To be more precise, network namespaces will only be effective when each process accesses its own virtual network device with a specific list of network filtering rules. Such construction would allow building extremely powerful sandboxes — for example it would be possible to restrict the list of IP addresses that the browser rendering code is allowed to connect to, and throttle the bandwidth available to services.

8.3. Authors' contributions

Authors of this thesis collaborated closely to create it, making it virtually impossible to distinguish their separate pieces of work. However, in order to give a better understanding of their involvement, below a list of their main responsibilities is presented.

Jędrzej Jabłoński was a main contributor to:

- testing,

- suEXEC module,
- requirements analysis,
- commenting the source code, documenting deployment,
- chapters 4, 5 and 7.

Marcin Pawłowski was a main contributor to:

- the C library,
- supervisor,
- Python wrapper,
- chapters 3 and 6.

Appendix A

Attached CD

In this chapter the content of the Compact Disc attached to this paper will be described. The disc has been recorded using ISO-9660 standard and contains the complete repository that we used to create our sandboxing tool. In the main directory the following files can be found:

1. *readme.txt* — text file describing installation process and demonstrating examples of usage,
2. *src/* — directory containing all the source files of our sandboxing tool,
3. *external/* — directory containing suEXEC patch, configuration files, and installation instructions,
4. *doc/* — directory containing all *.lyx documents and all figures that compile into this paper,
5. *tests/* — directory containing tools for testing our sandboxing tool.

A.1. Sandboxing solution source (*src/*)

Main modules of the sandboxing solution are as follows:

1. *api.c* + *api.h* — complete C interface,
2. *defines.h* — include file that contains hardcoded path to the supervisor binary,
3. *err.c* + *err.h* — error handling code,
4. *jail.c* — implementation of *start_jail,*
5. *jail.py* — commandline user's interface,
6. *prepare-*.sh* — init scripts that have to be executed before first usage of the sandbox (see: readme.txt),
7. *setuid_wrapper.c* — binary that executes jail.py Python script,
8. *supervisor.c* — supervisor code,
9. *sys_vfs.c* + *sys_vfs.h* — implementation of functions for VFS manipulation,
10. s*ys_methods.c* + *sys_methods.h* — implementation of other functions for permissions and resource manipulation.

A.2. Apache suEXEC module patch (*external/*)

Directory *external/* consists of the following files:

1. *suexec/readme.txt* — text file describing installation process of the suEXEC patch,
2. *suexec/suexec.patch* — the patch itself,
3. *suexec/test/index.cgi* — sample CGI script used in section (7.4),
4. *suexec/conf* — Apache configuration files used for testing

A.3. Documentation (*doc/*)

All LyX documents used in the process of composition of this paper are included in the *doc/* directory. Besides, all the diagrams in *.dia* and Encapsulation PostScript formats, and all plots in *.png* format are included in this directory.

Presentation *seminarium01.odp* (in Polish) was given by the authors at the Seminar on Distributed Systems on the 8th of December 2011 in Warsaw. File *jedrzej_jablonski_i_marcin_pawlowski.html* contains abstract of this presentation (also in Polish).

A.4. Tools used for testing (*tests/*)

The following programs and scripts were used as the base of the correctness and performance tests:

1. *allocate.c* — a program that allocates a given amount of memory,
2. *allocate2.c* — a program that allocates a given amount of memory after forking into two separate processes,
3. *cpu2.c* — a program that occupies a given CPU time after forking into two separate processes,
4. *readdev.c* — a program that tries to perform *open*, *read*, *seek*, *write*, and *close* operations on given file,
5. *empty.c* — an empty program — used to measure sandbox initialization time,
6. *cpu.sh* — simple script that occupies as much CPU as possible in a given number of threads,
7. *cpuperf.sh* — script used to perform CPU performance tests,
8. *vfs.sh* — script used to perform VFS performance tests.

Bibliography

[1] Ram Pai, Serge E. Hallyn, Applying mount namespaces, http://www.ibm.com/developerworks/linux/library/l-mount-namespaces/index.html — Tutorial on mount namespaces, their usage and limitations.

[2] Levy, Henry M., *Capability-Based Computer Systems, Digital Equipment Corporation 1984. ISBN 0-932376-22-3 — Paper introducing a concept of capability as a foundation of security in computer systems.

[3] Paul Manage, CGROUPS, http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt — Linux kernel documentation of cgroups.

[4] Simes, How to break out of chroot() jail, http://www.bpfh.net/simes/computing/chroot-break.html — Tutorial on how to break out of chroot VFS jail.

[5] http://penguinsecurity.net/wiki/index.php?title=How_to_break_out_of_a_chroot()_jail — Article describing methods of breaking out of chroot VFS jail.

[6] Limiting Resource Usage, http://www.gnu.org/s/hello/manual/libc/Limits-on-Resources.html — Documentation of rlimit mechanism as provided by Linux kernel.

[7] Linux man pages: clone(2), unshare(2), capabilities(7) — Documentation for Linux kernel syscalls.

[8] Linux process containers, http://lwn.net/Articles/236038/ — Article covering process containers for Linux (later renamed to cgroups).

[9] Overview of the seccomp sandbox, http://code.google.com/p/seccompsandbox/wiki/overview — Short overview of seccomp kernel mechanism.

[10] Alfred Spiessens: Patterns of Safe Collaboration, PhD thesis, Universit´ catholique de Louvaine, http://www.evoluware.eu/fsp_thesis.pdf — Section 8.1.5 contains a thorough explanation of “confused deputy” problem.

[11] Ferraiolo, D.F. and Kuhn, D.R. (October 1992), "Role-Based Access Control", 15th National Computer Security Conference. pp. 554-563 — Paper describing how RBAC system can be exploited to provide central management of security and why it is more suitable than DAC in security critical systems.

[12] Bell, David Elliott and LaPadula, Leonard J. (1973), Secure Computer Systems: Mathematical Foundations, MITRE Corporation — Paper providing one of the first strict definitions of security property of a system, as well as mathematical foundations for proving such property.

[13] SELinux documentation, http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5/html/Deployment_Guide/selg-overview.html#sec-acm-ov — SELinux documentation for Red Hat Enterprise Linux 5.

[14] suEXEC documentation, http://httpd.apache.org/docs/2.0/suexec.html — Documentation of suEXEC module for Httpd 2.x.