Graphical Security Sandbox For Linux Systems

Cosay Gurkay Topaktas

Dissertation 2014

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment
of the requirements for the
Erasmus Mundus MSc Dependable Software Systems

Head of Department : Dr Adam Winstanley

Supervisor : Prof. Barak Pearlmutter

June, 2014

DECLARATION

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of the others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:_____ Date:_____

Abstract

It has become extremely difficult to distinguish a benign application from a malicious one as the number of untrusted applications on the Internet increases rapidly every year. In this project, we develop a lightweight application confinement mechanism for Linux systems in order to aid most users to increase their confidence in various applications that they stumble upon and use on a daily basis. Developed sandboxing facility monitors a targeted application's activity and imposes restrictions on its access to operating system resources during its execution. Using a simple but expressive policy language, users are able to create security policies. During the course of the traced application's execution, sandboxing facility makes execution decisions according to the security policy specified and terminates the traced application if necessary. In the case of an activity that is not covered by the policy, the facility asks for user input through an user interface with a simple human readable format of the activity and uses that user input to make execution decisions and to improve the security policy. Our ultimate goal is to create a facility such that even casual users with minimal technical knowledge can use the tool without getting overwhelmed by it. We base our tool on system call interposition which has been a popular research area over the past fifteen years. Developed sandboxing facility offers an user-friendly, easy to use user-interface. It monitors the given application and detects activities that might possibly be system intrusions. Moreover, the tool offers logging and auditing mechanisms for post-execution analysis. We present our evaluation of the tool in terms of performance and overhead it generates when confining applications. We conclude that developed system is successful in detecting abnormal application activity according to specified security policies. It has been obtained that the tool adds a significant overhead to the target applications. However, this overhead does not pose usability issues as our target domain is personal use cases with small applications.

Contents

Chapter 1

Introduction

1.1 Computer Security

Computer security is a general term that covers a wide area of computing and it has become extremely important over the years as computers have become central for people and companies to conduct various businesses and to exchange very sensitive data using computers. Consequently, there are adversaries whose ultimate motive is to seize these data for evil purposes. The number of malicious applications on the Internet increases rapidly every year and locally occurred infections caused by malicious software has reached massive numbers in 2012[1] . Adversaries often disguise their malicious applications as benign and publish them on the internet in order to trick casual computer users into downloading and executing these applications locally, hence giving adversaries an opportunity to gain unauthorized access to system resources and seize or manipulate the sensitive data stored on users' computers.

Users usually need to download applications from time to time for simple tasks and they generally download these applications from unknown sources on the Internet after a brief search. They then execute these applications locally in order to do these small tasks and put their computers at risk as a result. Therefore,a mechanism that would monitor these applications and

[1] http://www.kaspersky.com/about/news/virus/2012/2012_by_the_numbers_Kaspersky_Lab_now_detects_200000_new_malicious_programs_every_day

warn users if they behave strange would be very useful, especially for casual computer users.

1.1.1 Threat Model

In order to embody what's being discussed, a simple real-world problem which everyone can face can be given as an example. Let's consider we want to convert a Microsoft Word file to Open Office format or *vice versa*. In order to complete this task, we download a small application from the Internet. If the website or the provider of the application is not known or does not seem reliable, we might get suspicious about the application behaviour. In order to increase our confidence in this piece of program, we could greatly benefit from a lightweight application confinement facility that can monitor the behaviour of this application to see if it behaves as expected. In this context, behaviour basically means the interaction of the application with the operating system, in other words system calls the application makes to the system. If the confinement facility finishes executing the program without a warning or termination,then we can be more comfortable when using this particular program. Cases like this example are where the motivation for such a monitoring system arises. Examples and different use cases can surely be enhanced.

As mentioned, computer security is a massive research area, thus we only focus on a small part of it. We're interested in the idea of application confinement and application confinement mechanisms. Application confinement mechanisms allow users to trace a targeted application activity to check if the application does not try to access or manipulate sensitive system resources.

Since we developed our facility for systems based on it, our domain in this project is Linux. We will talk about what kind of vulnerabilities an adversary can exploit in a Linux system in the next section.

1.2 Security in Linux

Unix systems, particularly Linux systems, have been known to be more protected and secure than Microsoft Windows and that malicious software for Linux is not as widespread as it is for Microsoft Windows. There are a number of reasons for this. First of all, the popularity of Linux has not been as high as of Microsoft Windows. Therefore, it has not been the first target of malware developers for years. Since the use of an operating system is directly correlated to the interest by the malware writers to develop malware for that operating system[2], the growth of use of Linux systems over the years consequently caused Linux malware to increase.

Secondly, Since Linux has multiple user environments with different privileges, a malicious software has to gain a high privilege to perform an action that causes a severe damage on the system.

The last but not least, the fact that Linux is open-source enables maintainers and developers to quickly patch a system vulnerability without having to deal with a company policy or bureaucracy which can take place in a company that develops a commercial operating system. In case of an exposed vulnerability, the system is patched quickly before malicious software infects other systems and gets wide-spread.

Having all aforementioned advantages, Linux too may exhibit vulnerabilities which can be used by an adversary. There a number of security attacks that can be performed towards Linux systems. As mentioned, a malicious program needs to gain a high privilege or root access to do a serious damage to the system. Therefore, gaining unauthorized access, usually through system calls, is an important type of attack. It is called privilege escalation. A root access gained adversary can easily traverse every part of the file system and manipulate sensitive system files or read confidential data like *rsa* keys or password information. Furthermore, previous study shows [12] that certain attacks, including network and file system related attacks, which can severely harm the operating system without needing root access can also be performed.

Attack scenarios diversify each year. Therefore it is important to monitor processes and their

[2]http://www.internetnews.com/dev-news/article.php/3601946

system call. The notion of process and system call in Linux will be detailed in the next section.

1.2.1 Processes and System Calls in Linux

In Linux and in Unix systems in general, an instance of a running program is called a *process* [11]. Every time a program is invoked, a process is created for that program and an unique id, which is a 16-bit number called *pid*, is assigned to that newly created process. A newly created process is also allocated a memory in which the executable code, program data, call stack and a heap resides. Furthermore descriptors of the system resources, which are called file descriptors in Unix systems, and set of security permissions and the processor state [2] are also allocated to the newly created process. A process may consist of multiple threads that execute simultaneously. Generally, processes have four different states that they can be in. These states are *ready*, *running*, *blocked* and *terminated*.

A system call is an interface between an user-space application and a service that the kernel provides[3] since a direct request to the kernel can not be made. This way, programmers are able to use kernel services through this interface. There are just over 300 system calls in Linux that can be grouped under five different categories, which are *process control*, *file management*, *device management*, *information maintenance* and *communication*.

Whenever a system call is invoked, that system call is multiplexed into the kernel through a single entry point. The *eax* register is used to identify the particular system call that should be invoked, which is specified in the C library. When the C library has loaded the system call index and any arguments, a software interrupt is invoked (interrupt 0x80), which results in execution (through the interrupt handler) of the *system call* function. This function handles all system calls, as identified by the contents of *eax* register. After a few simple tests, the actual system call is invoked using the *system call table* and index contained in *eax* register. Upon return from the system call, *syscall exit* is eventually reached, and a call to *resume userspace* transitions back to the user-space. Execution resumes in the C library, which then returns to the

[3] https://www.ibm.com/developerworks/linux/library/l-system-calls/

user application.[4]

1.2.2 System Intrusion and Sensitive System Calls

As many researches have been conducted in this area, system calls are very important as using them is the most fundamental way to perform a system intrusion. Previous work [3, 8, 16] discusses that the most sensitive system calls are the ones that are related to file system and network operations. These system calls, which manipulate file system, include but are not limited to, ***access***, ***open***, ***read***, ***write*** and system calls related to network operations are ***listen socket***, ***bind***, ***connect***. This is why, as it will be shown in the upcoming chapters, in our security tool, we focus on these system calls and their variants and ignore majority of the rest of the system calls in order to increase performance unless a system call is explicitly specified to be checked during the targeted application's execution.

In the next section, we will detail sandboxing and sandboxing mechanisms that can be employed to construct sandboxing tools.

1.3 Sandboxing

Sandboxing, also commonly called application confinement, is used to isolate and execute an untrusted application which is from an unknown source with certain restrictions enforced on what the application can do, in terms of accessing and manipulating sensitive system resources. Commercial anti-virus programs are a popularly used example of application sandboxing mechanisms. These anti-virus programs or sandboxes in general execute a piece of program and they warn users or immediately terminate the monitored program if the monitored program behaves in an unusual way and tries to access system sensitive system resources. By an *unusual* or *strange* way, we refer to the operations and system calls that might be outside of the targeted program's scope and are not necessarily needed for the program to perform what it is expected

[4] https://www.ibm.com/developerworks/linux/library/l-system-calls/

of it to perform.

There are different ways and approaches that can be employed to construct a sandbox mechanism. These approaches can be summarized as *seccomp*, *chroot*, virtualization and system call interposition. Virtualization and system call interposition are the two areas that many researches are focused on. We will examine these sandboxing types in the next section.

1.3.1 Sandboxing Mechanisms

seccomp, which is an abbreviation for *secure computing* is a built-in security mechanism in the Linux kernel. If *seccomp* mode is enabled, which is done using the ***prctl*** system call, targeted process is run in a secure state where its capabilities are very limited. Process running in *seccomp* mode can only make 4 system calls during its execution, which are ***read***, ***write***, ***exit*** and ***sigreturn*** and these system calls can be made to file descriptors which are available or granted to them. The targeted process is not allowed to make any other system call, else it is immediately terminated. The most important disadvantage of *seccomp* mode is consequently the fact that it does not allow any other system calls other than the four system calls mentioned above, which is very restrictive and will be very likely to have severe side effects on all targeted processes. There are some work done[5] to improve *seccomp* and these enhancements have already been adopted and used along with ***ptrace*** mechanism by some tools[10]. There are also some works[6] that are under way to extend *seccomp* mode to cover more system calls and enhance its capabilities. *seccomp* is also very dependent on the system architecture, thus making a mechanism on top it would be very unlikely to be portable and extensible.

chroot is a system call and a facility in Unix that is used to change the root directory of a running process and its children. When it is run using ***chroot***, a process sees the given directory as its root, thus cannot access or modify any file that is outside its given root. This is called *jailing* the running process. This operation of encapsulating an application in a modified environment is called *chroot jail*. ***chroot*** only works with file systems, thus it does not protect other sensitive

[5] http://lwn.net/Articles/475043/
[6] http://lwn.net/Articles/332974/

resources such as system devices or system network sockets. It also exhibits vulnerabilities that enable adversaries to escape the ***chroot** jail* environment easily by attaching itself to another process using ***ptrace*** or by performing a second ***chroot***[7], though some Unix systems have addressed this vulnerability in a fashion similar to `FreeBSD`. Another disadvantage of ***chroot*** is that it only root users can perform this command, making it impossible for non-root users to use a facility based on this. Therefore, it can be concluded that it is also not really suitable for building a sandboxing mechanism on top of ***chroot***.

Linux Security Model is another approach that has been adopted [6, 17] to construct security facilities. LSMs employ access controls lists to audit access attempts within a system. Frameworks such as *SELinux* and *AppArmor* are built-in LSMs in the linux kernel. As we will see in the next section, it does not align with our goals since it does not allow us to interactively generate security policies. Instead, security of the system is ensured through configuration of a static policy which can be cumbersome to specify.

Virtualization is considered to be the most secure and robust solution to construct an application sandboxing mechanism as there are many successful open-source and commercial examples such as `VMWare`[8] that prove this. Virtualization isolates the targeted program from the underlying operating system and its resources and runs it in a enclosed environment. Modifications made by the program in this environment are not permanently written on disk or registry, rather on a copy of them which are removed after the virtualization terminated. Therefore, the monitored program thinks it makes alterations to the actual system resources, whereas it makes alterations to the virtual version of these system resources, which are only to be deleted after the execution. Virtualization is a powerful mechanism to sandbox untrusted programs and is employed by many sandboxing programs such as `sandboxie`[9]. One disadvantage of virtualization of an operating system is that implementation of this would be very complex and difficult. In addition, It might be also possible to conclude that it is an *overkill* for monitoring small applications because we want to have a light-weight, easy to use security facility. Moreover, virtualization also has some performance and overhead issues which should not be

[7] http://www.bpfh.net/simes/computing/chroot-break.html
[8] http://www.vmware.com/
[9] http://www.sandboxie.com/

neglected, especially on computers with low specifications. The last but not least, we want to interactively generate security policies which can be later used when confining other applications with similar purposes. Virtualization can not provide a feature of this sort.

System call interposition (or interception) is an approach that has been used by a number of security tools [1, 3–5, 7, 10, 13, 15] over the years. System calls are the only way to access important kernel operations and since any definite change to an operating system is through system calls, many people have focused and worked on system call interposition and developed security tools based on it.

System call interposition based security tools usually use powerful system call tracing interfaces, which are ***ptrace*** in Linux and ***proc*** in Solaris OS. It will be detailed in upcoming chapters but to put it simply, ***ptrace*** interface allows a process to control and monitor another process. ***ptrace*** is primarily used for debugging applications such as `strace`[10]. System call interposition can be used to construct sandboxing mechanisms because it enables programmers to have total control over a process which is much needed to construct a security tool. Moreover, It allows us to construct a facility that is entirely in user space, with no modifications to the kernel. Therefore, it is more flexible and portable compared to other aforementioned security mechanisms, which is very valuable for us as portability of the tool is an important goal for us in this project.

We therefore proceed, in the next chapter, to discusses previous research and work conducted to construct secure sandboxing facilities, highlighting their commonalities and differences.

[10]http://man7.org/linux/man-pages/man1/strace.1.html

Chapter 2

Related Work

Since system call tracing and interception has been a popular area for many years, there are a number of system call interposition based security tools that have been developed over the years. We share common features and/or implementation details with some of these tools. We will detail these tools and explain why they would not align with our ultimate goals and how they differ in that regard.

2.1 Previous Tools

Even tough, there are some other system call interposition based security tools that have been proposed, the following are the closest tools to our area that we want to talk and reason about.

2.1.1 Janus

Janus is one of the earliest sandboxing mechanisms that is based on system call interposition. Therefore, fundamentally it functions in a similar way with our tool though there are significant differences. When a system call to be checked is invoked, Janus puts the traced process to sleep and checks its configuration file, can also be called security policy, to decide whether to allow

or deny the current system call. If the system call is denied, the sandboxing facility returns an error code to the traced process. In order to optimize the tool, they omit checking some system calls that operate on an open file descriptor such as ***write*** and ***read***, hence eliminate the overhead of switches between kernel and user space. Instead, they only regulate and check system calls such as ***open*** or ***dup***,that grant or manipulate file descriptors these system calls use. They do these type of checks for other system calls related to file descriptors as well for performance reasons.

There are also fundamental differences between Janus and our tool. They implemented their system using *proc* interface in Solaris OS since they argue that ***ptrace*** is vulnerable to race conditions and aborting system calls in using ***ptrace*** is cumbersome. Our system is implemented for Linux systems using ***ptrace*** interface. Janus has only one security configuration file for all processes, hence allows users to establish one global security policy whereas in our tool, policies can be multiple and a policy can be unique to a process. Furthermore, Janus does not resolve relative paths and check the system call accordingly. Instead, it aborts all system calls that try to access a resource using relative paths. Another important point is that *Janus* does not allow multi-threaded programs since it does not fully address vulnerabilities caused by race conditions. Later versions of Janus has multi-threaded support but does not fully address race conditions [3]. Lastly, Janus does not possess a logging mechanism for audit trailing/post-execution analysis, although it is discussed that it can be easily implemented [15]. All of these reasons make Janus an unsuitable sandboxing facility for our goal.

2.1.2 BlueBox

BlueBox is proposed and developed in IBM Research [8]. It also employs system call interposition to capture system calls and to enforce previously specified security rules, through use of the ***ptrace*** interface.

In their implementation, they use a similar technique to ours where they define and specify harmless system calls in order to make their tool more efficient in terms of performance as checking a system call obviously generates an overhead. Consequently, listing benign system

calls and skipping them directly introduces some performance improvements.

However, it has number of differences with our goal and our tool. First of all, it does not allow users to interactively generate security policies through a graphical interface, but rather employ static security policies that are set beforehand, which can leave an user with no chance to revert consequences of poorly written static policies. It is also only focused on access to file system resources, consequently omitting other significantly important network system calls such as ***socket***, ***bind*** or ***connect*** which may lead to an intrusion since these system calls, along with other system calls, can be used to perform a mimicry attack on the host system [12, 14]. In this regard, their tool seems to be more like a combination of system call interception and Access Control Lists(ACLs) of Linux where access permissions of a program or a file can be specified statically and restrictions are imposed upon violation of these specified lists. Lastly, they list system calls which they think to be harmless and not exposed to attacks. It should be noted that listed benign system calls can be subject to change by the users in our tool.

2.1.3 MapBox

Although *MapBox* is a system call based interposition tool, it works in a very different way regarding specifying and enforcing security rules. First of all, just like *Janus*, it is implemented for Solaris OS, thus it utilizes the system call interception interface(***proc***) of Solaris OS to trace and manipulate system calls. Another point where *MapBox* falls apart from our tool and what they do quite differently is that instead of having policy files for different processes, they classify behaviours of a number of applications and group the similar application behaviours together in order to impose security rules on each class of applications based on their expected functionality [1]. When a program is intended to run using *MapBox*, the user configures the policy file and assigns the targeted program one or multiple behaviour classes that specify which system resources the targeted program can access. For instance, if target application is marked with *reader* in MapBox, it means that it can only read certain files that are specified and it can not perform any write operations or network operations

They discuss that their confinement mechanism should require minimum user input and that

such mechanism should provide functionality that fits for all processes [1]. By doing that, applications in one specified class will reach the same system resources as the other applications in the same class. The notion of leaving user input out of the picture is somewhat against what we try to do in our security tool as we encourage user input through a graphical interface. Moreover, they face some substantial problems such as an application's resource requirements and/or very accurately grouping applications with similar functionalities in the same class. Lastly, as a side note, when the tool detects a system call that violates the security policy, *MapBox* rewrites arguments of the system call with dummy file names or path names and then allows the system call to proceed. This can be considered as a viable alternative to the routine of returning error codes to the system calls which other tools usually do.

2.1.4 Systrace

Among all the security tools, *Systrace* stands out as the most mature [3] and the closest sandboxing facility to our tool in terms of features such as interactively and iteratively security policy generation for each application, policy enforcement, intrusion prevention and a detailed logging mechanism. It's written by Niels Provols, who is now a distinguished engineer at Google[1] and has been working for Google for nearly eleven years.

Systrace tackles the problem of specifying robust security policies by employing an user interface and recording user input in a training session. This is partly what we want to accomplish in our tool. In our tool, users can specify policies without using the tool. Any activity that is not covered by the policy will be covered by the sandbox as it will record every user input and add to the security policy. Next, *Systrace* employs a hybrid architecture where some system call checks are made in the kernel and some inspections are made in the user space. They discuss that a hybrid architecture is successful in eliminating performance issues and provides efficiency in terms of overhead. Kernel hooks are used to provide a fast path for always permitted and/or always forbidden system calls [13]. If kernel is not able to use the fast path, then the system call information is exported to the user space daemon. User daemon checks for a

[1] http://research.google.com/pubs/author1.html

related security statement in the policy as to whether or not it will allow it. If there's no security statement which covers that particular system call, the graphical interface of the tool prompts users a dialog for their input as to how to treat that system call. At this point, users can either deny or permit that system call. The tool returns an error code the denied system call. Users are able to specify what type of error codes they want to return to the traced process in *Systrace*.

As similar as it is in terms of functionality, there are some differences and downsides of *Systrace* tool. First and foremost is that it does not work anymore with the latest kernels as its last major commit was nearly ten years ago. It also requires a boilerplate code to run which can be very tiresome for most users. Moreover, it is not used easily as it requires a training session for each application in order to exhaust all execution paths. At the end of this training session, the security policy is created for subsequent runs. Starting generating a security policy by directly asking users for input from very beginning of the execution of the traced process, raises some usability issues. Instead, allowing users to create a static policy first and then add statements interactively in the case of they are not covered by the policy is an alternative approach and this is what we chose to employ in our tool. The last but not least is that there are significant differences in the semantics of policy generation language and rules imposed between *Systrace* and our tool. We should also take into consideration the fact that it is not easy extend this tool as it's written in low-level C.

2.1.5 Ostia

It's been developed by the authors of *Janus* to eliminate maintainability and architectural security issues that were present in early versions of *Janus* [3]. It is considered to be a more secure tool among other system call interposition based security tools [16] since it employs a different and novel architectural approach, which is called a delegating architecture, instead of a traditional filtering architecture on which almost all of other works base their tools.

In a delegating architecture, unlike the filtering architecture that has been employed by earlier tools, tracer process works as an agent(medium) between the system and the targeted application. Most system calls are not delegated or checked as it has been discussed that only a

small number of all system calls can be used to perform a system intrusion [3, 9]. If the process makes a system call that needs to be delegated, Ostia converts this system call into inter-process communication message, and then delivers them to the agent which is the tracer process and after the security check is successful, the agent makes the system call on behalf of the targeted process and returns the result to this targeted process. This way, they aim to alleviate atomicity problems and race conditions [9] that were present in their former tool Janus. Ostia does support multi-threading as it creates an agent for each thread in an application. Since converting system calls into inter-process communication messages brings an additional overhead, *Ostia* delegates only a small number of system calls which they discuss to be the most sensitive system calls and permit the rest in order to alleviate performance issues. As previously mentioned, system calls such as ***open***, ***socket*** which acquire file descriptors and/or system calls such as ***bind***, ***dup*** which modify these descriptors are among these sensitive system calls.

They discuss that system call interposition based security tools that have a filtering architecture, where filtering means permitting or denying system calls in different cases, suffer architectural vulnerabilities like race conditions. They argue that race conditions are diverse and coming up with solutions to each and every one of these race conditions in piecemeal manner add significant code complexity. Delegated architecture based system, on the other hand fundamentally prevents these vulnerabilities by having a medium body between the operating system and the targeted application.

Like most sandboxing tools, Ostia does not have an interactive phase with the user in regards to generating security policies to be used for subsequent runs. Security policies are written after or in advance of a targeted program execution. Even though their policy language is relatively simple, poorly written policies can always pose risks for the operating system.

2.1.6 MBox

MBox [10] is a recent attempt at developing a sandbox mechanism based on system call interposition for preserving the integrity of the file system. It employs a novel approach by inserting a medium file system layer between the host file system layer and the operating system. After a

program is run using the sandbox, users are able to look at the changes made to the file system layer which is inserted by the sandbox and by looking at that medium layer, they can selectively make changes to their own file system. Inserting a medium file system layer is basically done with allocating a folder for the sandbox on the host file system. All the modifications the sandboxed program want to make recorded under this folder allocated for the sandbox. In order to redirect modifications that are intended to be made for the host file system, the sandbox intercepts system calls and rewrite their arguments to direct these system calls to the sandbox folder so that modifications will take place under that folder.

A significant feature they introduce is that they avoid race conditions by using a read-only memory in the traced process [10] whose possible applicability was also discussed in related work [9]. In order to be more precise, when a sensitive system call is invoked, at the time of inspection, the sandbox rewrites the argument of the system call with a pointer which points to a newly paged read-only memory in the traced process, then updates this read-only memory with the actual argument of the system call. This memory space is private and can not be modified. As a sandboxed program has to use a memory system call to change this read-only memory, the sandbox kills system calls regarding the memory such as ***mprotect***, ***mmap*** or ***mremap***. though they do not mention the side effects of failing these system calls. This is the approach we used in our tool as it seems to be a very reasonable and promising approach.

2.1.7 SELinux

SELinux is an implementation of mandatory access controls in Linux[2] and it is developed by the National Security Agency[3]. It has been integrated with modern kernels and come built-in with most popular distributions such as Ubuntu or Red Hat Enterprise. Even though its capabilities reach far beyond than that[4], SELinux can be used as a sandboxing facility through specification of static access control lists. It employs a type based enforcement mechanism, where a type can basically mean a file in your home directory or a running program. Most Linux distributions

[2]http://selinuxproject.org/page/FAQ
[3]http://www.nsa.gov/
[4]http://selinuxproject.org/page/AdvancedUsers

come with already configured default security policies of SELinux.

If running SELinux at all times is not desired by the user, enabling and disabling SELinux on your system every time you want to sandbox an application can be cumbersome. Moreover, configuring ACLs/policies are often difficult, especially for casual system users. In addition, it does not allow to interactively generate security policies which is one of the core features offered by our tool. It also does not provide a facility for users to trace application activity or detailed logging mechanism for post-execution analysis. It offers logging for operations that violate configured access control lists. Therefore, SELinux does not align with our ultimate goals as well.

2.1.8 AppArmor

AppArmor is regarded as an alternative and is similar to SELinux, which also enables users to create ACLs to protect their file systems and network devices. It is developed by using Linux Security Modules interface of the kernel. It is mainly focused on file paths. What it does additionally is that it offers a learning mode in which violations of a program are recorded and then using these records, new security profiles can be created. A number of Linux distributions Debian and Gentoo include AppArmor security system. It also suffers the same reasons such as lack of graphical interface or writing profiles being cumbersome as SELinux and hence we consider it to be not suitable for the main motive in this project.

2.2 Conclusion

In order to summarize, it can be well concluded that aforementioned tools lack some features which we would like to introduce in our system. Before talking about these points, first and foremost, we should note that almost all of these tools are obsolete and inoperative due to kernel changes over the years. Also, most of these tools are command based and lack a graphical user interface. Graphical user interface is something we want in our tool since we would like this

security tool be used even by casual computer users. More importantly, most of them do not have iteratively security policy generation in collaboration with the user. Semantics of policies and generation of these policies are another points where our tool falls apart from most of the other tools as other tools use technically deep security policies. As we discussed earlier, among other tools, *Systrace* is the closest work to our ultimate goal and It too is not maintained any more and does not work with modern Linux kernels without a patch. Also, its policy generation mechanism lacks some useful features. Additionally, it may be tiresome for users to use the facility as it asks for constant user input during the execution of an application. This is normally what we want, however there is no easy way to avoid user inputs related to irrelevant folders and paths since it tries exhaust all cases in its training session. Ostia, has the most secure architecture among them by putting a medium body between the operating system and the targeted application and by serializing the system calls of the traced process and by making these calls on behalf of that process. It does prevent most of the race conditions that filtering architectures exhibit.

In the next chapter, we will talk about our design and its implementation. We will talk about how we break down and address problems emerged in earlier tools and how we develop or fix features that were not present in these tools.

Chapter 3

Design

Before talking about the design and implementation, it is beneficial to summarize and remember our end goals in this project, so that our design choices become clearer and more meaningful later on. Ultimately, we aim to develop a graphical security sandbox for Linux systems that is both easy to configure and use, even for casual system users with minimal technical knowledge. In short, the security tool should have the following features:

- Unusual behaviour and/or system intrusion detection
- Expressive security policies
- Interactively security policy generation
- Ease-of-use
- Logging feature for post-execution analysis
- Flexibility
- Portability

Unusual behaviour detection: The security tool intercepts system calls that are made by the sandboxed application and consults with the security policy or policies specified beforehand. If there is a statement for a system call in the security policy and that system call violates that statement, it means there might be a system intrusion or the behaviour of the sandboxed program is unusual for the user who specified the policy. Therefore, that

system call will be denied the by the sandbox. If there are not any statements covering that system call, the sandbox asks the user about how it should treat to that system call. This way all uncovered activities according to the security policy are forced to be covered by the sandbox. This establishes a secure environment where abnormal activities are detected.

It should be noted that there is no formula or built-in support provided by the sandbox to the user in order to determine if there is a system intrusion by the targeted application since the notion of system intrusion can be seen as vague. Moreover, since it is intended for sandbox to be flexible, it is left out to the user to specify what the notion of unusual behaviour should be for a piece of program, though the sandbox enforces the specification of sound and correct security policies.

Expressive security policies: In order for sandbox to work correctly, it needs to be given sound and correct security policies. The tool recognizes a simple, yet expressive policy language for users to create meaningful and versatile security policies that are to be consulted with when sandboxing untrusted applications.

Interactively security policy generation: If there are no statements in the given security policy or policies that cover a system call, the sandbox gives the user a variety of actions regarding what it should do next. While the selected action is performed by the sandbox to treat the system call, the sandbox also converts this action into a security statement and improves the security policy by appending this newly generated statement at the end of policy file. This allows users to improve their policies interactively while increasing the restrictions on the targeted application.

Ease-of-use: Earlier tools suffer usability problems as users need to make an effort in the configuration phase. For example in *Systrace*, in order to run the tool, one has to write a relatively big chunk of boilerplate code which would not be possible for non-technical computer users. Furthermore, all these tools except for *Systsrace* are command based which again raises usability issues when sandboxing untrusted applications. In order to eliminate usability problems, as GUI programming stands out as a viable choice, we

develop a easy to use user interface which users only interact with during the sanboxing of untrusted applications.

Logging feature for post-execution analysis: The sandbox offers a logging facility so that users can log any activity of the programs that they run. A detailed logging of program executions can be very beneficial especially in the post-execution analysis phase to double-check if there has been an unusual activity performed by the untrusted application that the sandbox missed during its execution.

Flexibility: Since the sandbox is desired to be user friendly and not overwhelming, we leave everything from specification of safe and dangerous system calls to the specification of different levels of security policies and logging facility. Pre-configured restrictions or tool suggested features may contradict with the requests of the user, thus we have a sandbox where everything is up to the user to decide.

Portability: Most of the earlier developed tools are architecture dependent and modify the kernel, hence they on work systems in which they developed their security tools. We want our security tool to be portable, in other words, we want it to work on a variety of Linux platforms. That is why, as it will be detailed in implementation, we implement the tool in user-level as to make it able to work on other systems. However, there are some trade-off between portability and performance which we will talk about in the evaluation chapter.

Having discussed the features of our tool, we can now break down design and implementation details.

As briefly mentioned in *Portability* item above, we can employ different approaches to implement our tool. We can employ an approach that is entirely in kernel, we can employ an approach that is wholly in user-level or we have a hybrid approach where implementation is divided into both kernel and user-space. We eliminate kernel level implementation as it requires to modify the kernel heavily and is very unlikely to be portable though it is regarded to be better in terms of performance. Most tools employed a hybrid architecture in order to find a balance in performance and portability. However, that approach too requires some kernel modification

and with the rate of kernel changes and patches over the years, we now see that these tools either do not function anymore or need patches to work properly as they are partly architecture dependent. Moreover, kernel programming adds a significant complexity in the sandbox and a minor mistake in the modification of the kernel might lead to severe side effects on both the sandbox and the operating system.

We opted for an implementation that is entirely in the user-space, which allows us to make our tool portable and more compact. The cost of this choice could be having the overhead this approach can introduce since there is a switch between in kernel and user space in each time we intercept and inspect a system call. However, since we want this tool to be used in personal use cases which consist of sandboxing relatively small applications, performance concerns are considered as mere. In addition, we will talk more about performance with numerical results we obtained from running the sandbox with different programs in the evaluation chapter as mentioned earlier.

3.1 Usage Model

After starting up the sandbox by just typing *python sandbox.py*, there are couple of things users can do. They can either load their security policy file and edit it before starting sandboxing a program. They can also load and edit a file called *sys_calls* which is an auxiliary file that lists all system calls. Initially all of these listed system calls are marked with *permitted* which obviously means they are permitted system calls. Users can mark these system calls either with *lookup* or *forbidden* labels. The sandbox does not have to consult with security policy for system calls labeled with *permitted*. It should be noted that If there's a statement in the security file about a system call labeled with *permitted*, this statement is ignored and system call gets executed. System calls labeled with *lookup* are not automatically allowed to execute, instead the sandbox has to consult with the security policy file as to decide whether or not these system calls can executed. System calls labeled with *forbidden* are immediately denied without needing to check the security policy. Again, if there is a relevant statement regarding to a system call labeled with *forbidden*, this statement is ignored as well. Additionally, users

can mark a system call with *log* label which simply indicates that system call will be logged during the program execution. Figure 3.1 shows the overview of the usage model of the tool.

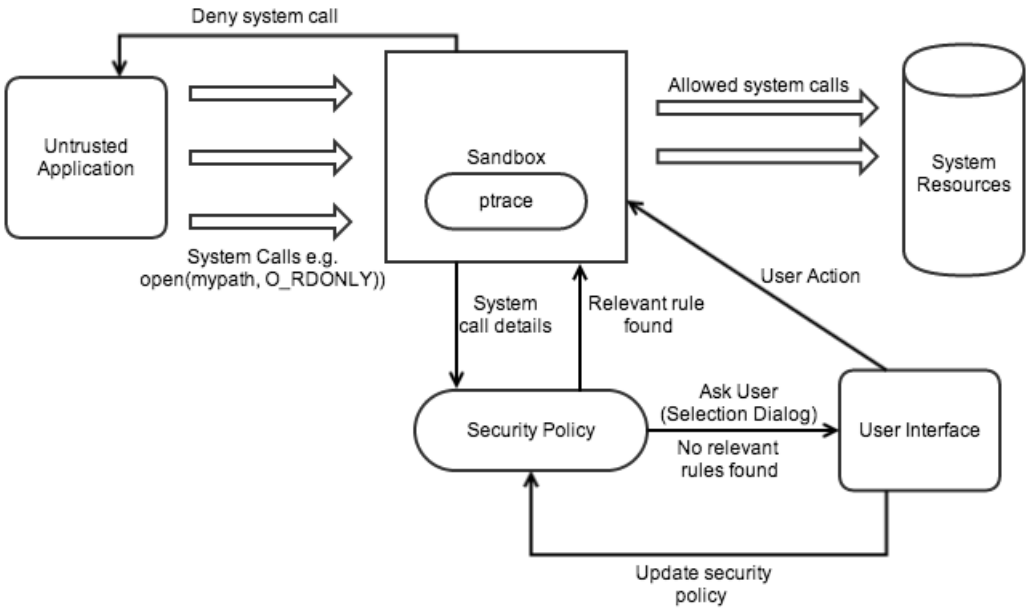

Figure 3.1: Usage Model of the Sandbox

Users can edit these aforementioned files either on the interface or in a desired text editor. The sandbox doesn't require *sys_calls* file or security policies to be a certain file type. The auxiliary *sys_calls* file is loaded automatically when the security policy is loaded into the interface. In the first run, the sandbox generates a *sys_calls* file in the same folder as the security policy file. For subsequent runs, users can either have the sandbox generate new *sys_calls* files and use them or they can use their existing *sys_calls* file, provided that security policy and *sys_calls* file are under the same folder.

After loading the security policy and the untrusted application to be sandboxed, users can start sandboxing the targeted application. The sandbox will invoke and get attached to the untrusted application using ***ptrace*** interface. Stopping at every system call, the sandbox will consult with the security policy if need be and look for a statement that covers the current system call. Depending on the security policy set, the sandbox will allow or deny the system call or kill the process altogether if necessary. If it can not find a relevant security statement, it will switch

back to the user interface and ask the user about how to proceed with the details of the system call. After user selects the action to be taken, the sandbox will again either allow or deny the system call or kill the process. The input of the user will be converted and appended at the end of the security file to be used for similar cases later on. Users also have the ability to deny system calls for one time only, and this action is not used to improve the security policy as it is a one time denial of a system call.

When the programs finish executing, either by terminating because of a security policy violation or reaching the end of the natural process, users can change the view on the user interface and take a look at the statistics of the targeted application's execution. All the system calls the targeted application made during its execution and denied system calls, if any, are displayed along with their number of recurrences and their frequencies. In another view, a visual plot which is constructed by using these statistics gathered during the program execution can be seen. Users are able to export these plots and statistics to a desired location. Lastly, system calls that are labeled with *log* in the system calls file are saved in a file called *system_calls_log* file under the same folder as the sandbox. Users can view these logs and perform post-execution analysis if they think to be necessary.

3.2 System Call Interposition

System call interposition is at the core of our design. We intercept and inspect many system calls of a sandboxed program and make execution decisions according to security policies. As mentioned, we use ***ptrace*** interface to establish a system that is based on system call interposition. Using ***ptrace*** to establish such a system is tricky as there a number of problems we need to address.

First of all, symbolic links may be used to refer to a resource in the operating system. Adversaries can use symbolic links to work around checks against security policies. For instance, users can specify a statement in the security policy that forbids the execution of a certain system call with a certain path. Using symbolic links or relative paths, an adversary can still refer to

the same system resource and avoid the check against the policy file since paths in the security policy don't match with the symbolic links given by the adversary. In order to eliminate this problem, we resolve all symbolic links and relative paths and gain the absolute paths of all files before checking them against the security policies. This way, there is no way of an adversary to make use of symbolic links since all links are converted to absolute paths.

Secondly, as it is discussed in related work [9, 13, 16], multi-threaded applications are not easy to check since they pose some risks regarding the system call arguments. When we inspect a system call and its arguments, after we check the system call arguments against the security policy but before the kernel actually executes it, another thread might come and rewrite these benign arguments in a manner to harm the system. Therefore it is important to eliminate this atomicity problem when dealing with multi-threaded programs. As mentioned earlier, there are possible solutions to this severe problem[3, 10]. The solution we employ to take on the problem is to page a read-only memory in memory space of the tracer process. Then what we do is to write sensitive system call arguments into this memory space and rewrite sensitive arguments in the system call with pointers that point to this read-only memory when inspecting the system call. This way, when the kernel reads the arguments to realize the system call, it will eventually read the arguments that reside in the read-only memory area of the tracer process. In other attempts [10], the traced process is to page a memory in its memory space and perform the aforementioned operation. Since newly created memory area belongs to the traced process, one key thing is to watch out for memory calls the traced process makes in order to update or disrupt this memory area. If there are memory system calls invoked by the traced process, these system calls are denied. Since this can have severe side effects on the traced process, we decided to page a private memory area in the memory of the tracer process which is out of reach of the traced process. This way, it is not required to deny all memory system calls while having a secure inspection mechanism.

Lastly, in terms of performance, switching back and forth between the kernel space and the user space to intercept and inspect system calls adds a significant cost which will be reviewed in the evaluation chapter.

3.3 Ptrace Interface

ptrace is an interface provided by Unix systems that allows a process, which is called the tracer, to control the execution of another process, called the traced process or the tracee. The tracer is able to monitor and intercept system calls that traced process makes to the system. Abilities of the tracer process range from manipulating the memory of the traced process and registers of the traced process to aborting the system calls or terminating the traced process altogether. ***ptrace*** is primarily used for debugging applications and constructing debuggers. There are widely available libraries such as `strace` and `ltrace` that are based on ***ptrace***.

ptrace takes four parameters as inputs which are *request*, *pid*, *addr*, *data*. *Request* represents the number of the operation ***ptrace*** will perform. There are number of operations[1] such as *peek data* or *poke data* ***ptrace*** performs. *pid* is the id associated with the process on which ***ptrace*** will perform operations. *addr* is the memory address to read from or write data to for ***ptrace*** operations. Lastly, *data* is the actual data to be written to the *addr* of the traced process. Some operations use *addr* and *data* parameters as other operations just ignore them. After performing an operation, ***ptrace*** returns a result code. 0 represents the success while -1 represents an error. We implemented our tool using Pyhton, however we now give examples of ***ptrace*** operations in C since it is a `libc` function.

In order to attach to a process, tracer makes the following system call:

```
long res = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
```

As seen, the process with *pid* becomes the traced process of the tracer, provided that ***ptrace*** is successful. Note that the last two parameters are not provided as it does not need or use them.

```
long res = ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
```

This statement allows us to continue and break at the next system call of the traced process. The traced process will stop when there is an entry into a system call. This is how we iterate on

[1] http://linux.die.net/man/2/ptrace

system calls throughout the execution of the traced process. As seen, it only takes the process id of the traced process and the request code representing the *sys_call* operation.

```
long data = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);
```

In this case, ***ptrace*** is called for *peek data* operation with the *addr* parameter which is the memory address used to read data from in the memory space of the traced process. One should be aware of the fact that return result might be -1 because the data read from the memory is actually -1, not because an error has occurred. Therefore the separation of the actual data and occurred errors should be taken into account in *peek data* operations.

```
long res = ptrace(PTRACE_POKEDATA, pid, addr, val);
```

poke data operation writes *val* parameter to the address space of the traced process with *pid*. Using *poke data* operation, the tracer can change values in the memory space of the traced process. In addition, *setregs* and *getregs* are similar operations provided by the ***ptrace***. They too are used to read from and write data to memory of the traced process. *poke data*, *peek data* and *setreg* and *getreg* can be used interchangeably depending on whichever is deemed to be more convenient for the user. In our implementation, we write and read from registers in order to manipulate values, thus we decided to use *setregs* and *getregs* operations as they are more convenient for us to use.

3.3.1 Ptrace-python Library

Since we chose to develop our tool in Python, which is a high-level language compared with C, so that the tool can be easy and straightforward to extend. Using Python, we can also make use of powerful libraries in the domain of GUI programming. We should also mention the massive support community it has that has evolved over the years. An important problem emerging from this choice is that ***ptrace*** is a native interface that is present in `libc` library of Unix systems. Therefore we can not use it directly the way it is done in C. That's where the ptrace-python library comes into the equation. It is a mature Python library which has been developed

for over five years now[2]. It binds Python to ***ptrace*** interface. How it manages to do this is that it ultimately loads and uses `libc` to bind ***ptrace*** interface with Python to construct this library. It loads the `libc`library through *ctypes* which *is a foreign function library for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It also can be used to wrap these libraries in pure Python*[3]. After loading `libc` through *ctypes*, it is possible to use ***ptrace*** mechanism through Python code. Additionally, it provides high-level API that can be used to perform various work such as debugging applications or analysing system calls of a process. We made minor modifications to this library so that when a system call is intercepted, it is directed to the sandbox for inspection. The below is a code snippet taken from the sandbox which shows how straightforward it is to attach an application to the sandbox.

```
tracer = PtraceDebugger()
tracer.traceExec()
tracer.traceFork()

tracer.addProcess(pid, True)
process = tracer[pid]
```

The first part of the code sets up the tracer which is created using *Debugger* module of the library. Second and third line enable the tracing of the ***fork*** and ***exec*** actions. It is important to keep these options enabled as the traced process might use them to create child processes. When these options are enabled, system calls of the created child of the traced process will also be traced and checked against the security policies. Therefore, the possibility of an adversary escaping the sandbox by creating child processes is eliminated. As it can be understood easily, the second part is the attachment of the process to the sandbox.

[2]https://pypi.python.org/pypi/python-ptrace/0.7/
[3]https://docs.python.org/2/library/ctypes.html

3.4 Security Policy

Security policies constitute the cornerstone of the tool as all intercepted system calls are checked against them. At this point, it should also be remembered that we leave specification of the security policies entirely to users as to have a flexible tool that they can freely use. Consequently, specification of security policies are extremely important. In order to have security policies that can be written in a high-level manner without getting overwhelmed by system specifics and are restrictive enough that the sandbox is effective on the targeted application, we develop a relatively simple but intuitive and rich grammar that users can look up to construct their security statements.

3.4.1 Grammar and Semantics

Grammar to be used for constructing statements have two distinct parts. One part is designated to be used by computer users with minimal technical knowledge so that they don't have to get too deep into technical specifics of the running process. The other part is for more advanced users who have an opinion about what is going on under the hood and can write statements covering technical details of running processes such as system call arguments. It is possible to have both type of statements in a security policy.

The part designated for casual users cover system calls that operate similarly or operate in the same action domain. The keyword to define a action domain is *operation*. If statement starts with *operation* keyword along with what type of domain it is, we group these system calls that are in the same action domain together and later check them against the policy. Instead of writing a statement for one system call with technical details, one can write a statement with *operation* keyword, thus cover a range of system calls. There are a number of high-level domains specified by the tool to be used by users.

$$operation : network$$

This is a sample operation statement which is not fully specified. The full statement is not shown since there are parts we will cover later in the section. In this sample, it can be seen that *operation* keyword is used with the domain network which is among the domain names specified by the sandbox. When the sandbox reads this statement, it knows that it should inspect system calls in the domain of network. System calls like *bind*, *connect* and *socket* can be given as examples for the system calls of the network domain. When a system call in the network domain is invoked, the sandbox will inspect this system call according to the rest of the security statement above. We mentioned these domains are predefined by the sandbox for casual system users and these predefined domains cover all of the sensitive system calls an user can be concerned about. These specified domains are as follows;

- network
- permission
- file name
- ownership
- directory
- input/output
- identification

A number of system calls are grouped under each of these domain names. We will now list these system calls and briefly explain what they are and why they are used. Network domain covers the following system calls:

shutdown, ***setsockopt***, ***listen***, ***bind***, ***connect***, ***socket***, ***accept***, ***sendmsg***, ***recvmsg***

These system calls are regarded as *sensitive* as an adversary can make use of them to disrupt the operations of other running processes or harm the system[3].

shutdown system call is used to close a open network connection.

setsockopt system call can be used change options of previously created sockets such as their domain or their protocol.

bind can be used to bind socket types, which might be not desired, to a certain port, which might be forbidden.

connect can be used to initiate a connection to an unknown host on an already open socket.

socket system call can be used to initially create a file descriptor to be used later for other network operations.

accept system call can be used to accept a connection created by an adversary.

sendmsg* and *recvmsg are also considered to be sensitive because they can be used to exchange file descriptors between processes which leads to indirect access to system resources [9].

Permission domain covers the following system calls:

chmod, ***fchmod***, ***fchmodat***

chmod can be used to change permission of a file to reduce its restrictions on who can and write that file which is highly undesirable for confidential files in the file system.

fchmod* and *fchmodat are used in the same way as ***chmod*** except that ***fchmod*** takes file descriptor as input instead of the file path and ***fchmodat*** takes file descriptor along with the relative path to that file descriptor as input.

Filepath domain covers the following system calls:

readlink, ***readlinkat***, ***link***, ***linkat***, ***rename***

readlink can be used to read a symbolic link created by an adversary for a forbidden path.

readlinkat is similar as the ***readlink*** system call except that it takes a file descriptor and a relative path to that file descriptor as input.

link can be used to create an alias for an existing file.

linkat is similar as *link* except that it takes a file descriptor and a relative path to that file

descriptor as input.

rename is an important system call as it can be used to simply change the name or the location of a file.

Ownership domain covers the following system calls:

chown, ***fchown***, ***lchown***, ***fchownat***

chown is used to change the ownership of a file.

fchown is the same as *chown* system call except that it takes a file descriptor instead of a path as input.

lchown is also identical with ***chown*** except that it does not resolve symbolic links.

fchownat is identical to the system calls above and it takes file descriptors and relative paths to these descriptors as input.

Directory domain covers the following system calls:

chdir, ***fchdir***, ***chroot***

chdir system can be used change the directory of a process which can be exploited by an adversary [9].

fchdir is a similar system call as ***chdir*** that takes the file descriptor as input. As mentioned earlier in *seccomp*, ***chroot*** is used to change the root directory that the process sees.

Input/output domain covers the following system calls:

open, ***openat***, ***creat***, ***ioctl***, ***close***, ***truncate***, ***ftruncate***

open **and** ***creat*** are very important system calls as they are used to create file descriptors and files which is a fundamental operation in a file system. That's why they have to be guarded very carefully.

openat is identical to these system calls above except that it takes a relative path as input.

ioctl system call is very diverse and can be used to control and manipulate many system devices.

close system call can be used to close a file descriptor which can be used to disrupt another process trying to read the file of that descriptor.

truncate system call is used to truncate a file to a given length which again can be used to disrupt system files.

ftruncate is the same as ***truncate*** system call except that it takes file descriptors instead of file paths as input.

Identification domain covers the following system calls:

setuid, ***seteuid***, ***setgid*** , ***setegid***

All of above system calls are used to change the user id, which can be user, group or both, of a process. These system calls can be used to have a process run with higher privileges than of users who runs it.

Having detailed system calls, we can now proceed to how to construct security statements using these domains and system calls. As discussed, we develop a simple, high-level security policy grammar to write security statements. The grammar itself is divided into two, which are for network operations and file system operations. Grammar for network operations is as follows:

```
$type: $opname: permit $except $arg $input
$type: $opname: permit $except $arg $input and $arg $input
$type: $opname: permit $except $arg $input and $arg $input
$type: $opname: permit $arg $input
$type: $opname: permit $arg $input and $arg $input
$type: $opname: permit $arg $input and $arg $input and $arg $input
$type: $opname: permit $except $general
```

It is possible to construct different types of statements as can be seen above. Statements are obviously parametrized and take multiple variables. Parameters of the variable *type* are as

follows;

$$\langle type \rangle ::= \langle operation \rangle \mid \langle system\_call \rangle$$

When *operation* keyword is used, the sandbox will then expect to see a domain name such as network. *system call* keyword declares that a specific system call will be the input of the statement. Second parametrized variable is *opname*;

$$\langle opname \rangle ::= \text{network} \mid \text{bind} \mid \text{connect} \mid \text{accept} \mid \text{listen} \mid \text{sendmsg} \mid \text{recvmsg} \mid \text{shutdown}$$

When *network* parameter is used, the sandbox will cover all of the system calls above whereas if a specific system call is given, the sandbox will cover only that system call.

Next, it is to decide how restrictive security statement is wanted to be. There are two approaches that can be taken. Before talking about these approaches, parameters of the variable *except* can be listed;

$$\langle except\_params \rangle ::= \text{all except} \langle exceptions \rangle^{+} \mid \text{null}$$

First approach is to construct a statement using *all except* which states that the sandbox should permit every transaction of the given system call except for the arguments that are written next. This approach can be useful If there are specific arguments that users want to cover and allow the same system call with other arguments to proceed automatically. It is possible to construct multiple security statements with *all except* related to a single system call. These statements will be treated as if they were *i*n a single statement by the sandbox. Second approach is to construct statements without the *all except* and with *permit arg input* keywords. In this kind of statements, only the given input is permitted by the sandbox, thus it will ask for user input for other arguments of the same system call. Clear distinction between writing security statements using with and without *all except* is that the sandbox won't ask for user input for system calls covered with statements with *all except* since statements state that all system calls should be permitted except the given arguments. If a security statement without *all except* is constructed for a system call with specific arguments, the sandbox will ask for user input provided that the same system call is invoked with different arguments since there are no statements covering the arguments which the sandbox inspects for the first time.

If an user is concerned about, say, only one network connection and does not want to be asked decision questions about how to proceed by the sandbox, writing statements with *all except* would be much preferable for that user. This distinction will become clearer when some example are given later on.

The next keyword is *arg* which is used to define the name of the argument of a system call. Parameters of this variable are as follows:

$$\langle arg \rangle ::= \text{type} \mid \text{protocol} \mid \text{sockaddr} \mid \text{domain} \mid \text{port} \mid \text{sockfd} \mid \text{socket} \mid \text{msg}$$

Some parameters of $\langle arg \rangle$ variable such as *sockfd*, *sockaddr* can be used by more advanced computer users who has an idea about how underlying mechanism works while some such as *protocol*, *port* can be used by more casual users. For listing connections and their arguments, we make use of an external library which is handy for network operations [4]

We also generalized the connection types for casual system users. Instead of writing detailed security statements which can overwhelm them, they can simply use *general* variable to construct simple, yet effective security statements. Parameters of *general* variable are the following:

$$\langle general \rangle ::= \text{incoming connections} \mid \text{outgoing connections}$$

As it is self-explanatory, users can construct security statements that can permit or forbid outgoing or incoming connections being made to the system. The sandbox determines whether a connection is outgoing or incoming and then checks it against the security policy.

Last variable is *input* which complements the *arg* variable. It represents values of arguments of a system call. Inputs can vary, so following parameters are only small sample of what parameters of *input* variable can take:

$$\langle input \rangle ::= \text{sockfd} \mid \text{AF\_INET} \mid \text{TCP}$$

[4] https://github.com/giampaolo/psutil

sockfd is the socket descriptor while *AF_INET* describes the domain, *IPV4* in this case, to be used by a socket and TCP which describes the protocol to be used. Following are a few examples that can be written using network grammar:

```
system_call: socket: permit domain AF_INET
system_call: socket: permit all except type SOCK_STREAM
system_call: bind: permit socketaddr 74.125.224.72
operation: network: permit incoming connections
operation: network: permit all except outgoing connections
```

The second part of the grammar is for file system operations. While grammar for file system operations are similar than of network operations, there are some differences. Grammar to construct security statements covering file system operations are as follows;

```
$type: $opname: permit $abspath
$type: $opname: permit all under $abspath
$type: $opname: permit all except $abspath
$type: $opname: permit all under $abspath except $abspath
$type: $opname: permit all under $abspath with $ext
$type: $opname: permit all under $abspath except $abspath with $ext
$type: $opname: permit all under $abspath with $ext $abspath except
   $abspath
$type: $opname: permit all except with $ext
$type: $opname and $arg = $input: permit all with $ext
$type: $opname and $arg = $input: permit all with $ext except
   $abspath
```

Variables *type* and *opname* are similar to those of network grammar. *opname* in this grammar represents domains *permission, file name, ownership, directory, input/output*. The first difference between two grammars is that argument and inputs are given before the actual security policy in this grammar. The reason for this is that goal of a system call related to file system can depend on its input values. There are modes and flags and other inputs that change capabilities of a system call. Therefore, if we want to specify input values in a statement, as they can

be omitted, we start by declaring those input values after the *type* and *opname* variables. *arg* and *input* variables are similar to those of network grammar. Following are a small sample of parameters these variables can take:

$\langle arg \rangle, \langle input \rangle$ ::= mode|fd|flags|null|O_RDONLY|O_WRONLY|SIWUSR|O_NOFOLLOW|null

The most fundamental statement that can be written is permitting a file name or file path.

The sandbox supports the use of global environment variables and relative paths in the security policy as it resolves all paths before performing an operation on the system call if necessary. Some examples for paths are as follows;

$$/cosay/Desktop/myfolder \mid etc/passwd \mid HOME/.ssh \mid ../myfile$$

Security statements related to file system operations are diverse and can be constructed in different ways. First, it is possible to permit every folder and file under a folder using *all under* keyword. It is also possible to permit everything except for a single file or a folder and its contents using *all except* keyword. We can express exceptions such as a file or a folder under permitted folders. Moreover, it is possible to make use of file extensions with *ext* keyword when constructing security statements. We can append *ext* keyword to every policy statement as it can be seen above. Variables of *ext* keyword can be any extension since it is up to users to define these security statements. Some examples are as follows;

$$py \mid pdf \mid doc \mid java \mid rsa \mid cs \mid class$$

We can construct statements for permitting a particular file type or permitting every file except a particular file type or we can construct statements for permitting a file type under a certain folder or permitting a folder except for a file type. There are many variations that users can do when writing these statements. Some security statements covering file system operations are as follow;

```
operation: filename: permit all under /cosay/Desktop/
system call: open: permit all under /cosay/Desktop/ except with .py
```

```
system call: read: permit all except with .ssh
system call: read: permit all except /proc/
```

The important difference between *permit all under* statement or *permit path* and *permit all except* should be noted here as well since system calls covered with security statements that are constructed using *permit all except* will not be asked to the user. Therefore, one should be careful when using *permit all except* or any other statements that include *except* variable. Statements constructed using *under* variable are more flexible as uncovered arguments will be asked to the user.

Last but not least, users are also able to include security policies in security policies when using the tool with the keyword *include*. For instance, once an user has a complete security policy which covers system files, They then can use this file as a base policy in other policies before adding application specific statements for various applications.

The following example shows how to add another security policy in a security policy:

$$include\ base\_policy$$

If the given path is not an absolute path, then the sandbox will look for the included policy in the same folder as the security policy file. If the path is absolute, then the policy will directly add this base policy. In the next part, we detail how we can generate new statements through the use of user interface.

3.4.2 Security Policy Generation

In order for the sandbox to work properly, it requires to be given a security policy by the user which is logical since it will otherwise ask for user input for every single system call it handles as there are no statements in the policy that cover these system calls. We previously discussed that in order to provide a fast path for the sandbox and specify logging option for system calls, we have a separate file called *sys_calls* file. As soon as a security policy is loaded

into the sandbox, it checks if there are any existing system calls file in the same folder as the security policy file. If there is one, it will load that into the sandbox as well so that users can see and make modifications on it. If it is the first time for users to run the sandbox, the sandbox generates a new *sys_calls* file under the same folder as the security policy file and loads it automatically into the sandbox. Initially, all the previously mentioned system calls are marked with *permitted* keyword so that users can specify which system calls the sandbox should inspect. Obviously casual users would not know well enough which system calls they should mark with *lookup* so that the sandbox will inspect them. Therefore security statements that include domain names are more powerful than statements that include an individual system call. if an user defines a security statement in the policy that includes a domain name, system calls under this domain will be inspected without a need to mark them with *lookup* in the *sys_calls* file while all the statements that include individual system calls will be ignored. In order to use security statements that cover individual system calls, users have to mark each system call in the *sys_calls* file with *lookup* keyword. A sample of system calls in the *sys calls* file are below;

```
socket: lookup - log
ioctl: permitted - log
open: lookup - log:detail
access: permitted - log:name:res
```

Using system calls above, if an user for instance writes a security statement that includes ***ioctl*** system call in the security policy, that statement will be ignored since it is marked with permitted in the *sys_calls* file. However, if an user writes a statement that includes domain input/output, then ***ioctl*** system call will be inspected. There are also *log* keyword which allows an user to log system calls. When the security policy is loaded and *sys_calls* file is generated and configured, one can start running the sandbox. The sandbox asks for user input in case of it does not find a relevant security statement for a system call. When users select an action from a number of options, before performing the relevant action, the sandbox converts this action into a security statement and appends it to the security policy so that next time it won't have to ask for user input as the rule which covers that case has been generated and appended to the

security policy. This way, security policies are improved throughout the execution of a targeted application. For instance, a few of the options the sandbox presents an user for a file system operation are the following:

```
Always permit the system call open except this input
   /home/cosay/.ssh/mykey.rsa
Always permit the system call open except this folder
   /home/cosay/.ssh/
Always permit the system call read for inputs of this type .rsa
```

If the user selects the first option, the following security statement will be generated and appended to the security policy:

$$system\ call:\ open:\ permit\ all\ except\ /home/cosay/.ssh/mykey.rsa$$

If the user selects the second option, the following security statement will be generated and appended to the security policy:

$$system\ call:\ open:\ permit\ all\ except\ /home/cosay/.ssh/$$

If the user selects the third option, the following security statement will be generated and appended to the security policy:

$$system\ call:\ read:\ permit\ all\ except\ .rsa$$

The next time a case like this takes places, users will not have to face a selection from the sandbox as the sandbox will decide how to proceed automatically. It should also be noted that options presented to the user are dynamically created by the sandbox depending on the system call and its arguments. For example, if the current system call is related to network operations, then the options related to *folder* or *input types* above will be changed with newly generated network options and arguments depending on which system call it precisely is. Changed options can be a port number or a domain type or any other argument related to network operations.

Since security policies with grammatically wrong statements would pose security vulnerabilities which otherwise wouldn't be present with well-defined security policies, it's extremely important to properly construct these security statements. In order to aid users to define grammatically correct security statements, the sandbox performs a static check of the security policy file before starts executing the traced process. Should there be an invalid security statement, it does not start the execution and raises a warning to the user with the line number of the invalid statement. This way, users can check again and correct their invalid security statements. When all statements are correct, the sandbox starts the execution of the given program.

3.5 User Interface

User interface should be easy to use and very intuitive so that even first time users won't have trouble running and using the tool. Therefore we have a simple, userfriendly interface which packs five different views, with each of which serving different purposes; *Sandbox view*, *System calls view*, *Security policy view*, *Statistical view* and *Graphical statistics view*. As seen in Figure 3.2, Sandbox view is where the user has most of the interaction with the tool. Security policy and program to be sandboxed can be selected either by the *File* menu above or using keyboard shortcuts. *Control+o* is used to load the untrusted application and *Control+p* is used to load the security policy. As a side note, keyboard shortcuts are not case sensitive so that users can use the tool with Caps Lock on. After loading both files, users can start running the sandbox by clicking on *Start Sandboxing* button and then see the system calls made by the application in the *System Calls* area. After they are finished, users can quit the application again either by using the *File* menu or using the *Control+q* shortcut.

When the security policy is selected, the sandbox loads security policy and *sys_calls* file into the interface. *System Calls tab* is used to view and edit *sys_call* file and *Security Policy* tab is used to view and edit security policies.

When there is an activity that is not covered by the security policy, the sandbox will ask for user input as seen in Figure 3.3 and use the answer to improve the security policy. When a

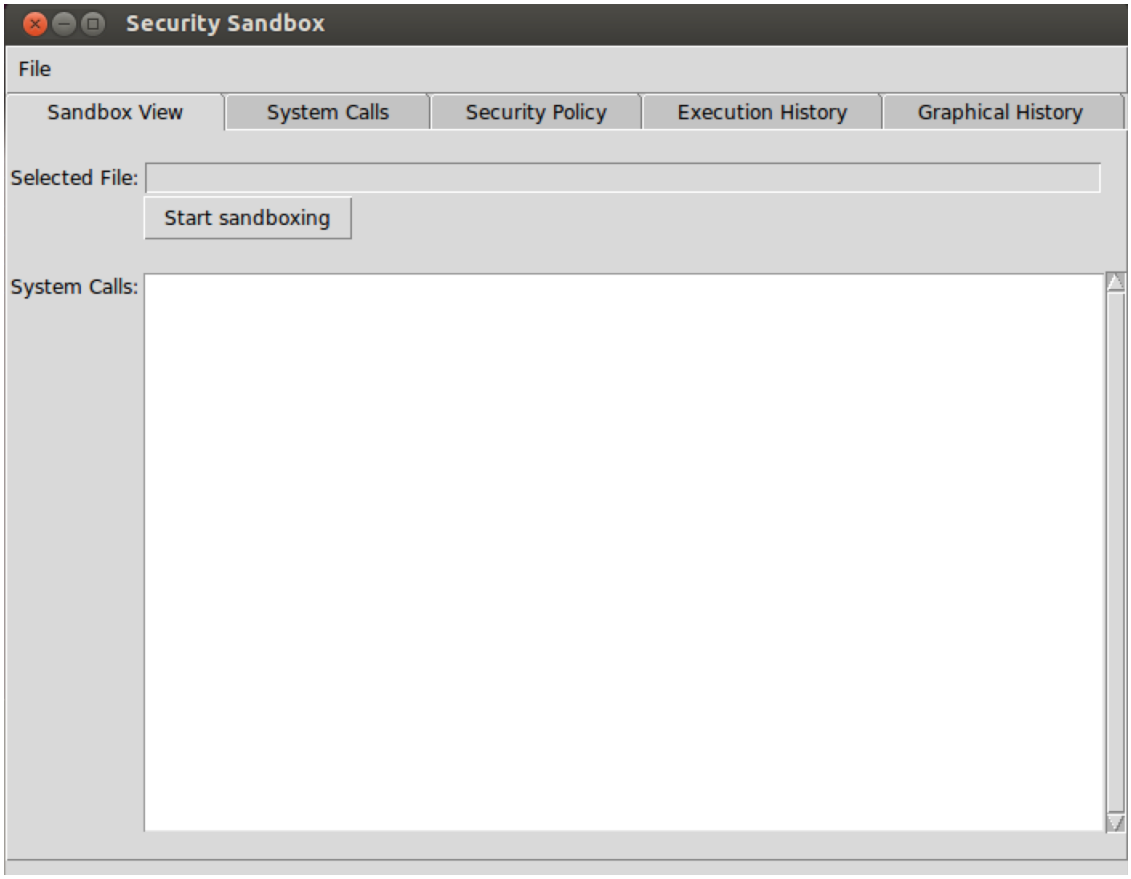

Figure 3.2: User Interface of The Sandbox

violation of the security policy takes place, as seen in Figure 3.4, the sandbox asks users one last time whether or not they want to proceed to denying the system call. This is asked since side effects of denying a system call can be severe for the targeted application. In addition, poorly written security policies might lead to denying a benign system call which is highly undesirable. Therefore, asking users one last time also allows them to review what system call is about to be denied, thus giving users more control over the tool. At this point, they can cancel denial of a system call.

3.5.1 Logging Facility

Post-execution analysis of a targeted application can be needed sometimes to manually ensure that the sandbox worked correctly and no harmful system calls have been made by the targeted application. For this purpose, the sandbox offers a logging facility to users. As discussed

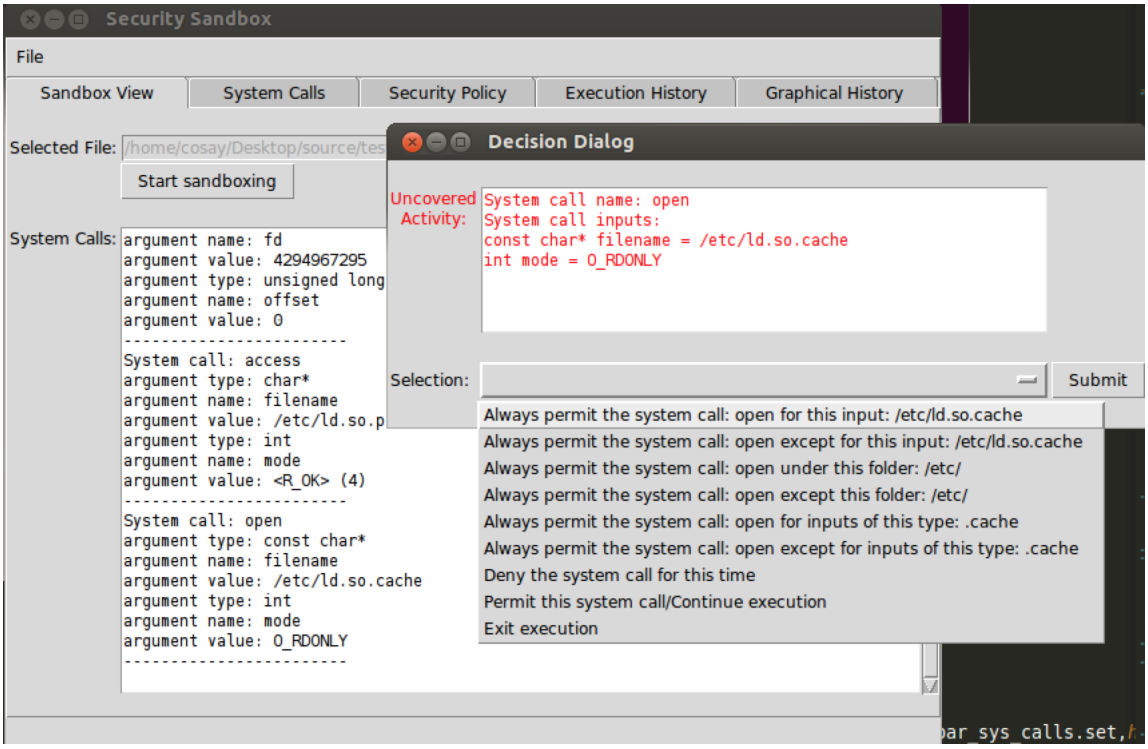

Figure 3.3: Selection Dialog

previously, users who want to log system calls can simply mark the system calls in *sys_calls* file with *log*. Also, level of logging can be done in two different ways. If users are not interested in the specifics of the system call, they can simply mark the system call with *log:name* which will tell the tool the record only names of the system calls. If users wish to learn more about the system call such as its inputs, they can mark the system call with *log:detail* which will give them a more detailed interpretation of the system calls. Furthermore, users can also record the result of each system call. In order to log the result returned from each system call, users can mark system calls with *log:name:res* or *log:detail:res* depending on the log level desired. System calls are logged with the exact time(microseconds) in which they were executed. At the end of program execution, marked system calls can be found in a file called *sys_call_log*

3.5.2 Statistical Dashboard

In some cases, users with some understanding of the underlying system might want to see what the targeted application has done during its execution. Information such as what system calls

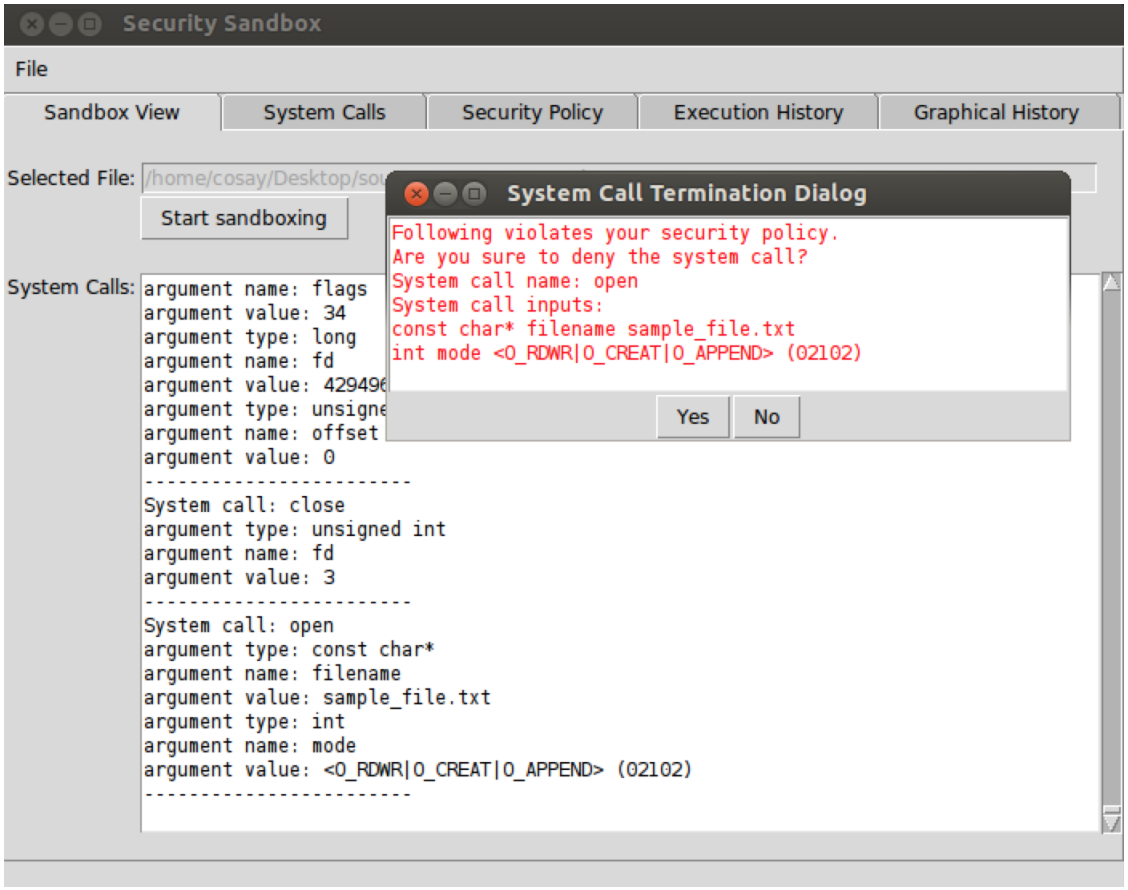

Figure 3.4: System Call Termination Dialog

have been made, how many distinct system calls or how many system calls in total have been made along with which system calls are denied and frequency of system calls can be computed by the sandbox during the execution. After the execution, this information is provided to interested users in the *Execution History* tab in the interface. This provides users a more detailed insight into the targeted application's execution flow. Moreover, in the *Graphical Execution History* tab, the visualised version of the execution information can be found. As seen in Figure 3.5, both statistical execution history and plotted execution history can be exported in a desired file format.

In the next chapter, we will present our evaluation of the tool and talk about results we have obtained from experimenting with the tool.

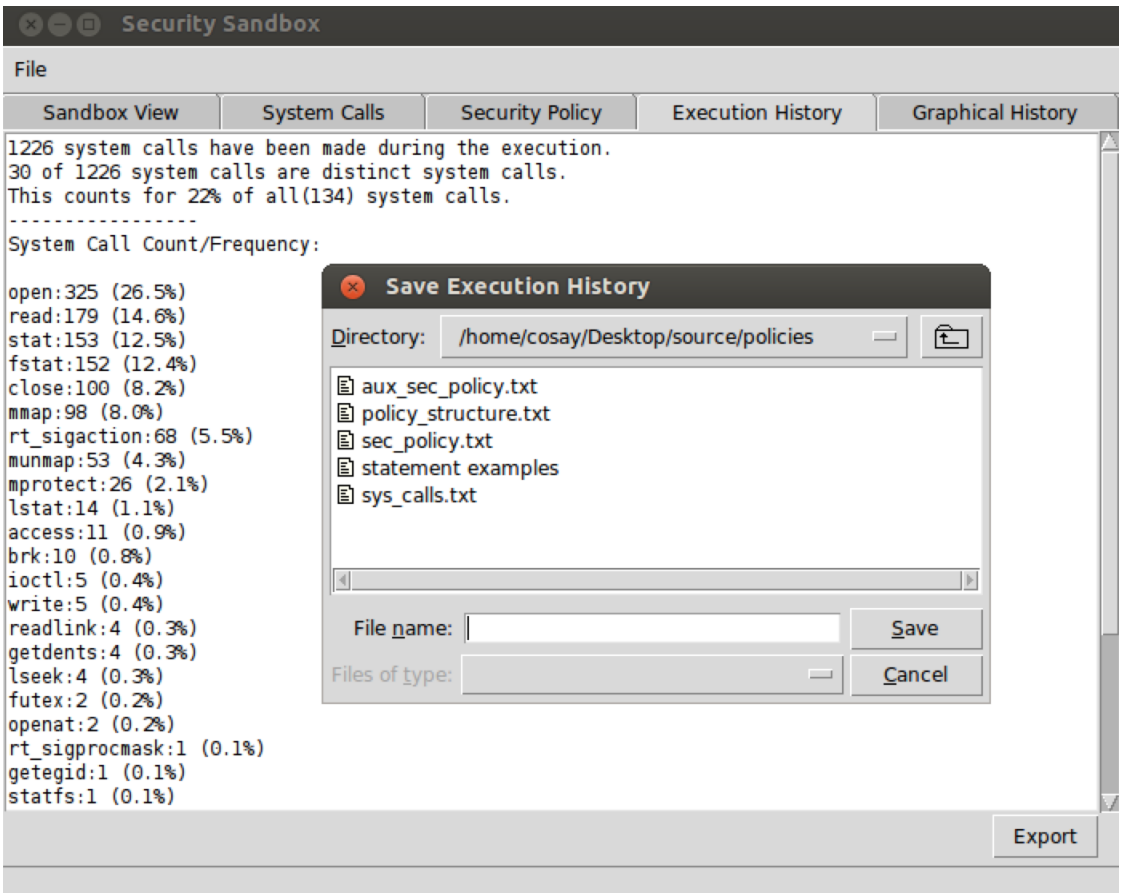

Figure 3.5: Execution History Export Dialog

Chapter 4

Evaluation

We evaluate our tool in terms of performance and overhead it introduces to the targeted application and compare our results with related work. We also talk about performance issues and bottlenecks the sandbox exhibits due to various reasons. In addition, we also make a sound security analysis of the tool and examine its effectiveness. Last but not least, important concepts such as portability and compatibility are covered with regards to the sandbox as well.

4.1 Performance and Overhead

One of the significant aspects of a security tool is to be efficient in terms of the overhead it introduces to the runtime of the traced application. Performance evaluations have also been made in previous work. Therefore we will compare our results with the results obtained in previous work.

We measured our tool's performance by testing it with small custom applications that emulate common utility applications such as file input/output or network applications with having abnormal activities such as trying to access a forbidden folder or open a connection.

Our tests were carried out on a Ubuntu 12.04 LTS instance of `VMWare` with 4 GB of ram

and dual core 2.5 ghz CPU. For each application, we ran our tool ten times and recorded the execution times. In addition, we also computed the standard deviation in order to be more precise about the data. Table below shows results we obtained from running the sandbox two different applications which respectively perform various file system and network operations.

2.890 Benchmark	Without Sandbox	With Sandbox	Increase
IOApp	298ms $\pm$ 34.8ms	2.890 ms $\pm$ 219ms	9694%
SocketApp	17.1ms $\pm$ 2.25ms	106ms $\pm$ 12ms	628%

Table 4.1: Overhead on applications

It can seen that the sandbox adds a huge overhead to the targeted application. Before talking about its reasons, it should be noted that even with this significant overhead the sandbox generates, applications work in a reasonable amount of time without any lags, which is tolerable for small applications. For more complex applications, this can become a serious concern but as discussed, confinement of complex applications is not within domain of this project. More professional application confinement mechanisms like anti-viruses or virtualization software can be used for more complex applications.

One of the major reasons why the tool exhibits a significant overhead is that it constantly makes context switches between the kernel space where the system call is intercepted with *ptrace* and user space where the inspection of system calls take place. Moreover, we canonicalize every symbolic link and relative path which are then kept in private memory area to avoid race conditions. Furthermore, denying system calls have been performed in our tool through manipulation of the system call name with a safe system call. Consequently, this unexpected action from the operating system leads processes to make more system calls which take additional time. Lastly, the fact that the sandbox asks for user input is another point to increase the time in user space.

Lastly, we present a comparison of our tool with the previously proposed security tools in the below table. As a side note, the results of other tools are directly taken from their studies. The previous tools have been tested on system call intensive applications

4.1.1 Comparison With Existing Tools

Benchmark	Janus	Ostia	Systrace	MapBox	Our Tool
SysCallIntensiveApp	8%	25%	31%	41%	969%

Table 4.2: Comparison with Other Tools

As seen, the overhead each application confinement mechanism generates are significantly different, with our tool being the slowest one. Design choices and end goals contribute to the creation of this table as mentioned earlier. All the tools in the table modify the kernel to implement their mechanisms and it should be remembered that kernel checks provide faster execution since there are no context switches between kernel and user space. Thus this results in previous tools having faster execution times compared with our tool which is implemented wholly in user space. In addition, as there are supporting cases[1][2], we also suspect that the language used in the implementation might slow down the sandbox since it is written in a high-level language like Python, and unlike compared tools all of which are written in C. Unlike previous tools, we also use medium libraries to realize system operations which might be another factor to the performance of the sandbox.

We now proceed to talk about the effectiveness of the tool which is another important aspect of application confinement mechanisms.

4.2 Effectiveness of The Tool

The core feature of a sandboxing mechanism is, obviously, its ability to find abnormal activity of a targeted application. However, effectiveness of a security tool is not tangible or measurable like performance or overhead as the notion of abnormal activity varies greatly from an application to application. Furthermore, as we discussed in the previous chapters, the definition of abnormal activity of a program is specified in the security policy by the user. For instance, accessing a particular system folder might be considered as abnormal activity, thus a violation

[1] http://onlyjob.blogspot.ie/2011/03/perl5-python-ruby-php-c-c-lua-tcl.html
[2] http://theunixgeek.blogspot.ie/2008/09/c-vs-python-speed.html

of the security policy whereas accessing the same file can be regarded as benign in another security policy. Therefore, effectiveness of the tool can ultimately be decided by the user. In addition, it is not possible to conclude that a tool is always or generally successful in detecting abnormal activity as the sandbox might not be tested against many and a range of computer applications. In our tests, the tool managed to detect what we specified to be abnormal activities. Alternatively, in order to prove that the tool is conceptually effective, we will conduct a detailed security analysis on different security concepts and vulnerabilities and propose possible solutions as it is a standardized approach also adopted previous related work[1, 3, 10, 13]

4.2.1 Security Analysis

System call interposition based tools face different problems[9] due to the underlying *ptrace* mechanism which they use. Therefore, different approaches and architectures have been proposed to solve common problems in system call interposition based tools[9, 10]. The most important problem that needs to be addressed is race conditions. There are different types of race conditions that an adversary can exploit within a system. All these race conditions are present in execution of multi-threaded programs as they can perform TOCTOU(time of check, time of use)[3] attacks. We now list these race conditions that were previously studied[3] and give more details about them.

Relative Path Race conditions

Race conditions emerged from the use of relative paths take place because of the fact that a thread can change the actual path between the resolution of the path by the sandbox and the resolution of the path by the kernel. An adversary can perform an attack using exploiting this race condition. The following scenario can be given as example: Let's consider we have the following statement in our security policy about system call ***open***:

system call: open: permit all except home/user/

[3]http://cwe.mitre.org/data/definitions/367.html

Now, let's consider we have thread A of an application which resides under

home/user/Desktop/myfolder/appfolder

that makes a open system call with the path

../../somefile

The sandbox resolves this relative path and get the absolute path of

home/user/Desktop/somefile

and consults with the security policy. Security policy sees that it's not under *user*, so tells the sandbox to proceed with the system call. Now, let's consider thread B of the same application which makes the system call ***rename*** with the following arguments

rename(home/user/Desktop/myfolder/appfolder, home/user/Desktop/myfolder)

and changes the path. When the kernel gets the ***open*** system call and resolves its path, it will have the path

home/user/

which is a violating path since security policy forbids a ***open*** system call with this path. As seen, an adversary manages to work around the sandbox and can read the contents of the folder *user* by exploiting the relative path race condition.

Symbolic Link Race conditions

Symbolic links are important since an existing one can be changed or new ones can be created by an adversary to use at any given time. For this case, we can use the scenario above by slightly changing it. We have the same security statement in our policy as above and thread A makes the same system call with the same path. The sandbox checks the system call against the security policy and allows the system call to proceed. Now, assume thread B of the targeted

application makes a ***link*** system call and creates a new symbolic link

home/user/Desktop/somefile

for the following path

home/user/

and the kernel reads this newly created symbolic link

home/user/Desktop/somefile

which points to the path

home/user/

and realizes the ***open*** system call. As seen, adversary once again manages to work around the sandbox and reads the contents of the folder that was specified as forbidden in our security policy. The partial solution[9] to this is to deny creation of symbolic links for paths that are forbidden security policies. This can work with new symbolic links but exploit of existing symbolic links is still possible, thus needs further examination.

Working Directory Races

As threads of an application share the same working directory, an adversary can exploit this race to perform an attack. Again, our first scenario can be used for this case as well. We have the same security statement and assume the ***open*** system call made by thread A has an argument of file name e.g. *somefile* and the sandbox allows this system call since current working directory of the application is *appfolder*. Now, at this point, thread B can a ***chdir*** system call with the path

/dev/

change the path. The new path becomes

/dev/somefile

and kernel processes this system call, thus allows an adversary to read the contents of the forbidden folder and the security once again has been violated.

Proposed Solutions

Previous examples show us that race conditions are a serious problem for application confinement as they can be used by an adversary to escape the sandbox. In our design, race conditions have been attempted to be eliminated by canonicalizing the relative paths and rewriting the argument in the system call with the absolute path. Canonicalizing relative paths is not sufficient because it does not resolve the concurrency problem as another thread still can rename the application folder and rewrite system call arguments after the inspection by the sandbox. We address this problem and other race conditions by using another approach that was proposed [3] and successfully adopted [10] previously. In this approach we first page a private memory in the address space of the tracer process which can not be modified by any parties other than the owner of the memory. Then we initialize a pointer that points to this memory address. Lastly, we update this memory address with the system call argument and replace system call argument with the pointer we just created. Since an adversary can not modify this memory space, they can not perform any attacks. As mentioned, this approach was used before and found useful [10], but our approaches differ in practice. We page a memory space in the tracer process which is immune to modifications by the traced process while other attempts [10] force the traced application to make a ***mmap*** to page a memory in the memory space of the traced process. Since the traced process owns this process, it can modify this memory space. Therefore, the traced process is not allowed make system calls such as ***unmap*** or ***mprotect*** in order to change this memory address. Conceptually, this approach enables our tool to avoid race conditions.Our solution to symbolic links have also been discussed previously.

4.3 Portability

We develop this tool entirely in user space with no modifications to the kernel as portability is one of our main goals in this project. While an user-level approach might be slower, it however has the benefit of having no dependencies on the system. Only dependency of the tool is the *Tkinter* module as the graphical interface has been realized using this particular Python module. Since our tool is system independent and does not need kernel patches as a requirement, it can be used without a complex installation which might be tiresome for most users. In order to test if our tool works on other platforms, we installed four major Linux distributions and ran our tool on these platforms. The installed platforms are

- Ubuntu 12.04.4 LTS
- Debian GNU/Linux Release 7.5(wheezy)
- Fedora Release 20 (Heisenberg)
- Linux Mint Release 17 (Cinnamon)

As expected, our tool worked without any problems on these platforms which increases our confidence to conclude that it can be used on other Linux distributions as well.

4.4 Compatibility

A sandboxing mechanism also needs to be compatible with various programs so that it can actually be beneficial for personal use cases. Therefore we ran our tool with a range of programs and concluded that our tool is applicable with a variety of programs. Our tool can be used to sandbox executables, Python programs, Java programs and C programs. In addition, it should be noted that our tool sandboxes programs without patching or modifying them in any way. One disadvantage of the tool is that, since it is used through an user interface, it does not support command line utilities or programs.

Chapter 5

Conclusion

In this project, our ultimate goal was to implement a secure, light-weight application confinement mechanism to aid Linux system users to increase their confidence in programs that they use every day. We have carefully examined different security mechanisms provided by the Linux kernel and decided to use *ptrace* interface of the Linux, which enables a process to manipulate and have total control of another process, kernel to construct our application confinement facility. We have also examined previous attempts at implementing an application confinement mechanism in order to adopt some of their features and avoid common mistakes. Examining related work helped us to identify key points and problems in constructing an application confinement mechanism. These key points can be categorized as security policy generation, detection of abnormal activity specified by the security policy, ease of use and race conditions, performance and portability. We have addressed security policy generation through three major steps. First, we have implemented an easy-to-use user interface which involves users in interactively creation of new security statements which are added to security policies. Next, we have designed a simple, yet expressive policy language which can be used by both technical and more casual users to construct sound security policies. Lastly, we have implemented a system which iteratively generates security statements from user input to improve security policies. In order to ask for user input, we translate complex arguments such as file pointers or file descriptors into understandable formats.

Detection of abnormal activity is performed by checking system call and its arguments against security policies specified by the users. In our trial runs, we have seen the sandbox facility successfully detects forbidden activities defined in security policies.

Unlike most of the previous work, this project has an user-friendly interface which dramatically decreases the effort to correctly use such security mechanism. Moreover, the effort to specify a security policy is minimal as it can be done interactively using the user interface.

Race conditions are important problem for most application confinement mechanisms. Multi-threaded programs can escape the sandbox through exploiting time of check and time of use vulnerability. Multi-threaded programs can change arguments of a system call after they are inspected by the sandbox but before they are actually executed by the kernel. We have conceptually addressed this problem by adopting an approach that was found to be useful in a recent work [10]. We page a private memory in the memory space of the sandbox and update this memory with system call arguments at the time of inspection. Then we create a pointer that points to this private memory area and rewrite the system call arguments with this pointer. This way, a traced process can not disrupt the memory space of the sandbox, which eliminates the problem of argument races.

We have evaluated our application confinement mechanism in terms of performance and compared the results obtained with the related work. We have seen that our tool adds a significant overhead to the targeted application. This overhead is also much greater than the related work. We conclude that this is direct result of a design choice since we have implemented our facility completely in user level, with no modifications to the Linux kernel as to have a portable tool. As existing tools have a hybrid architecture in which they modify the kernel to gain advantages in terms of performance, the overhead they generate is smaller than ours.

The choice of implementing the system entirely in user level allows us to have a tool that is portable across different Linux systems. We have tested our tool on a number of Linux distributions and obtained that it works on these systems without any problems, as the tool has no system dependencies. It's also very straightforward to run and use the tool.

It can be concluded that we achieve our end aforementioned goals in this project. However, de-

sign choices forced us to have some shortcomings. We have some performance issues but this is balanced with the portability of the tool. The tool also exhibits some application compatibility issues because it's not possible to work with command line utilities with an user interface. Most importantly, security sandboxes that have been implemented using system call interposition approach have become obsolete and inoperative due to the fact that kernels have evolved and changed over the years whereas our tool works well with the modern kernels and will continue to work as it is not system dependent. Only condition that can affect its operability is the possible deprecation of system calls in the future.

As much as it presents the documentation of the work that has been done, this dissertation also gives insights about sandboxing mechanisms in general and previous research conducted in the area of application confinement.

We will explain possible future work that is needed to improve our application confinement mechanism.

5.1 Limitations and Future Work

As the acclaimed computer scientist Edsger W. Dijkstra have said

> *Testing shows the presence, not the absence of bugs*

Testing never ensures correctness of a piece of software. Therefore, even though it has been heavily tested, our tool still needs more testing and it needs to be run with more software to be regarded as *stable*. In addition, although the resulting implementation have met our end goals, it exhibits some shortcomings that need further research and work. First and foremost, the we eliminated race conditions and made a security analysis of our tool conceptually. We haven't been able to create a case where multiple threads perform a time-of-check-time-of-use attack. The related work does not also mention if their tools have been tested against a multi-threaded malicious software which attempts to exploit race conditions. Therefore elimination of race conditions needs further study.

We have also mentioned our that tool suffers some serious performance issues. Although we are not too concerned about performance since the target domain of the tool has always been personal use cases with small applications where performance issues do not cause usability problems, further work can be conducted in order to decrease the overhead our tool generates when confining applications.

As much as our tool is compatible with executables and a range of other programs, it does not work with command-line programs and utilities. Possible future work can be done in this domain to ensure that the tool supports all kinds of applications.

Lastly, privilege elevation is very important since it can be exploited by adversaries to gain unprivileged access. This is why our tool does not allow privilege elevation of a process which is not useful and efficient. *Systrace* for instance allows privilege elevation on a system call basis. System calls gain privilege for their operations and these privileges are immediately dropped after the system call is executed. Not all of related work refers to this problem. Future work is needed to deal with privilege elevation as to make our tool more efficient.

Bibliography

[1] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th Conference on USENIX Security Symposium—Volume 9*, SSYM'00, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251306.1251307.

[2] Abraham Cagne, Greg Galvin, and Peter Baer. *Operating system concepts with Java*. John Wiley and Sons, 6th edition, 2004.

[3] Garfinkel, Pfaff, and Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of Network and Distributed Systems Security Symposium*, NDSS 04, San Diego, CA, 2004.

[4] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. Slic: An extensibility system for commodity operating systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268256.1268260.

[5] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography—Volume 6*, SSYM'96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267569.1267570.

[6] Serge E. Hallyn and Phil Kearns. Domain and type enforcement for linux. In *Proceed-*

ings of the 4th Annual Linux Showcase & Conference - Volume 4, ALS'00, pages 15–15, Berkeley, CA, USA, 2000. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268379.1268394.

[7] *User-Level Infrastructure for System Call Interposition: A Platform for Intrustion Detection and Confinement*, ISOC Symposium on Network and Distributed System Security, San Diego, CA, USA, February 2000. ISOC.

[8] *A Policy-driven, Host-Based Intrusion Detection System*, ISOC Symposium on Network and Distributed System Security, San Diego, CA, USA, February 2002. ISOC.

[9] *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*, ISOC Symposium on Network and Distributed System Security, San Diego, CA, USA, February 2003. ISOC.

[10] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 139–144, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2535461.2535478.

[11] Mark Mitchell, Jeffrey Oldham, and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, 1st edition, June 2001.

[12] Chetan Parampalli, R. Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 156–167, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-979-1. doi: 10.1145/1368310.1368334.

[13] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium—Volume 12*, SSYM'03, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251353.1251371.

[14] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications*

Security, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: 10.1145/586110.586145.

[15] David A. Wagner. Janus: An approach for confinement of untrusted applications. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1999.

[16] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, pages 2:1–2:8, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1323276.1323278.

[17] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-00-5. URL http://dl.acm.org/citation.cfm?id=647253.720287.