Proposed Model for Sandboxing in Linux

Palak Thadeshwar, Rohan Vora, Aishwarya Ramachandran, Lakshmi Kurup

Abstract*— The proliferation and popularity of the Internet has led to average Internet users downloading various utilities and applications from the Internet very frequently. Often, these applications are downloaded from untrusted users and websites, or from unverified third parties and suppliers. Due to this, it has become very important for a casual user to differentiate between a malicious and a benign application. This has become excessively difficult because of the rise in number of malicious applications on the Internet. In computer Security, Sandboxing is a mechanism that allows unknown or untrusted code into the system, and yet does not let it damage the system. A sandbox isolates the running program from the rest of the system by imposing restrictions on network resources and file system access, and keeps the host system safe. A sandbox system heavily restricts the program from inspecting the host or reading from the input device. In this paper, we review existing tools that provide sandboxing mechanisms. We compare what features have been used by each, and highlight the advantages and disadvantages of each. In the end, we propose a system that will incorporate the best features of these tools, yet be user-friendly.

***Index Terms*—Computer Security, Sandboxing, Seccomp-bpf, System call interposition.**

I. INTRODUCTION

Computers are being used on a large scale nowadays not only for personal use but also for commercial purposes. In the process, a lot of sensitive data is being exchanged using these computers. This has also led to a rise in the number of users trying to steal data in order to use it for malicious purposes. As a result, it is extremely necessary to secure computers and the data contained in it from such attacks. The amount of harmful applications available on the Internet is humongous and recent study shows that the widespread infections caused by these applications is increasing rapidly with every passing year. The attackers often make their malicious applications look harmless before publishing them online so as to trick naïve computer users into downloading and executing them on their system locally. Once such applications are executed on the local system, the attacker gains unrestricted and unauthorized access to the user's system. The attacker can now seize or modify sensitive data and also access the system's resources. In order to avoid such attacks and to increase the user's confidence in the fidelity of the application, an application confinement mechanism can be used. It must monitor the behaviour of the software to verify that it works as expected.

Revised Version Manuscript Received on November 06, 2015.

Palak Thadeshwar, Department of Computer Science, Dwarkadas J. Sanghvi College of Engineering, (Maharashtra). India.

Rohan Vora, Department of Computer Science, Dwarkadas J. Sanghvi College of Engineering, (Maharashtra). India.

Aishwarya Ramachandran, Department of Computer Science, Dwarkadas J. Sanghvi College of Engineering, (Maharashtra). India.

Prof. Lakshmi Kurup, Department of Computer Science, Dwarkadas J. Sanghvi College of Engineering, Mumbai, (Maharashtra). India.

A. Security in Linux

It is believed that Linux systems are more protected and secure than Microsoft Windows. It is also a known fact that a very small amount of malicious code has been developed to compromise Linux systems whereas the code written to compromise a Windows system is extensive. There are a number of legitimate reasons for this. Linux systems have never been as popular or widely used as Windows systems have. As a result, attackers have targeted Windows systems over the years for the simple reason that the infection will affect a large amount of users. Also, Linux systems support multiple user environments wherein different users have different access rights and privileges with respect to the system. As a result, a piece of malicious code has to gain high privileges to perform actions that may severely damage the system. Lastly, as Linux is an open-source operating system, developers can quickly patch a vulnerability in the system much before it causes any serious harm to it or spreads to other systems.

However, in spite of all the above-mentioned advantages, a Linux system too, can be compromised by an adversary. A classic example of a security attack on a Linux system is gaining unauthorized access to it through system calls. This is only possible if the adversary's software gains root access to the system. Once the adversary has gained root access to the system, he can not only traverse the file system, but also read and manipulate sensitive, confidential data contained in the system. Certain network and file system related attacks can also be performed without needing root access to the system. As the popularity of Linux systems is on the rise, the number of malware being developed to compromise them is also increasing.

B. Processes and System Calls

A running program in Linux or Unix is called a process. Each process is given a unique 16-bit number called pid. The process is also allocated a set of security permissions and the processor state [2]. At any point in its lifetime, a process can be in one of the following four states- ready, running, blocked and terminated. System calls are extremely important for system intrusion. System calls are highly essential as user applications cannot make direct requests to the kernel. A system call is an interface that acts as a bridge between the user and a service that the kernel provides. The most sensitive system calls are the ones related to file system and network system. The most basic system calls related to file system are access, open, read and write but are not limited to the same. The basic system calls related to network operations are listen socket, bind and connect. If the system calls that the application can make are not restricted,

then it can access and manipulate sensitive system resources.

C. System call interposition in Linux

System call interposition is a technique which has been used since a long time. In sandboxing, system call interposition is used because it helps the programmer to have total control over a process. Various system call interposition methods are:

- **Ptrace**: Ptrace, an abbreviation of "process trace", is a system call using which a parent process can not only observe the behaviour, but also control the execution of another process. The parent process can examine the core image and registers of another process and also change it. This mechanism is primarily used to perform system call tracing as well as breakpoint debugging.
- **Proc**: On Solaris, you can use /proc; /proc lets you specify the subset of system calls that you are interested in wrapping, which lets you achieve better performance at the cost of compatibility.
- **Ld_preload**: LD_PRELOAD is a dynamic linker flag which allows any other shared library to be loaded before any other library. If you set LD_PRELOAD to the path of a shared object, that file will be loaded before any other library (including the C runtime, libc.so). So to run ls with your special malloc() implementation, do this:
  $ LD_PRELOAD=/path/to/my/malloc.so /bin/ls

D. Sandbox systems

Misuse of system calls to compromise a system is a potential threat and to avoid it, sandboxing is used. In computer Security, Sandboxing is a mechanism that allows unknown or untrusted code into the system, and yet does not let it damage the system. A sandbox isolates the running program from the rest of the system by imposing restrictions on network resources and file system access, and keeps the host system safe. A sandbox system heavily restricts the program from inspecting the host or reading from the input device. Sandboxes can be used to analyse applications or test their legitimacy by allowing them to execute in a highly controlled environment so that they cannot cause any harm to the rest of the system. A sandbox can restrict the operations of a code by allowing it only as many permissions as required, without granting additional permissions that can be used to harm the system. Thus, sandboxing provides security by isolating a software so as to prevent malicious code from abusing the system. This can be done by specifying security policies for the applications or isolating each application in a separate virtual machine.

II. SANDBOX FEATURES

A. Security Policy

Security policies provide a description of the system calls that can be made by and the other privileges given to an application. The sandbox system intercepts any system calls made by the application and compares it with the specified policies. If the application is allowed to make the system calls that it is trying to make, then it will be allowed to proceed. However, if a system call made violates the specified policy, then there can be a possibility of system intrusion. In such a case, the sandbox system denies the system calls as they might produce behaviour that is unusual or harmful for the host system. These security policies play a very important role in the operation of the sandbox system and helps the host system to be protected from malware.

B. System call interposition method

Tools in Linux used for filtering system calls. This can be done using one of the methods from among ptrace/proc/ld_preload as described above.

C. System calls check

Sandbox system usually checks a few system calls which are more important to the goal of achieving security. A few tools interpose all system calls which results in overhead, consequently causing performance issues. A few tools interpose a few system calls causing less overhead, but is less strict, and hence may be less secure.

D. File system changes

Sandbox tools may either choose to allow read and write system calls which makes direct changes to the files , or it may disallow those system calls altogether. Some sandbox tools allow changes to virtual files, which may be reflected in the actual system if the user permits.

E. Logging

Post-execution analysis of a targeted application might be useful 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 system may offer a logging facility to users.

III. RELATED WORK

Sandboxing using system call interposition has been popular for a long time now and thus a number of security tools based on system call interposition have been developed. In the following section we will review some of these tools by sharing their features.

A. Janus

Janus is a sandboxing tool which mainly focuses on system call interposition. When a process makes a system call, Janus puts it to sleep and confirms with the security policy of the process whether it can be allowed to proceed [4]. After checking the security policy, Janus can either allow or deny the system call. In case system call is denied, then the tool returns error message to traced process. This tool checks only those system calls that grant or manipulate file descriptors. It does not check system calls such as write and read as they operate on already open file descriptors. As a result, the overhead caused due to switching between the kernel and user space is reduced. Janus is implemented using proc interface in Solaris OS. It allows user to specify only one global security policy and the same configuration file is used for all processes.

The main drawback of Janus is that it does not allow multi-threaded programs as it cannot handle race conditions. However, the newer version supports multi-threading. Apart from multi-threading, Janus also does not have logging mechanism. So if the user wants to analyse the behaviour of a program after its execution, then the user cannot use Janus.

B. BlueBox

Blue Box uses ptrace interface to intercept system calls and impose the specified security rule [6]. It generates a list of harmless system calls and skips these system calls at the time of interposition. It supports static security policies that need to be defined previously. It does not allow user to specify the policies dynamically. It supports interposition of system calls for filesystem resources. However apart from the file system related calls there are various system calls that affect the system like network.

C. MapBox

MapBox is also based on system call interposition but its operation in different from the tools that we have seen previously. Like Janus, MapBox too uses the proc interface of Solaris OS for tracing and intercepting system calls. However, it uses a slightly different method for specifying security policies. Mapbox creates classes based on the behaviour and the functionality of program [3]. It specifies a separate security policy for each class. When the program is running in the sandbox, it is assigned one or more behaviour classes and based on the policy file assigned to these classes, the program is permitted to access the resources. The major drawback of the MapBox is that multiple programs belonging to the same behaviour class will be given same access rights which is not reliable. When the security policy is violated MapBox allows the program to make the system call but ensures that the changes are made to a copy of the original file. Thus the host system is not affected and the running program is unaware of the where the changes are being made.

D. Systrace

It not only has a detailed logging mechanism but also supports generation of different security policies for different applications. It allows the user to specify the policies dynamically via an interface. The system calls in this tool are partly checked in the kernel and partly in use space [5]. If the system call is known to be harmful or harmless for sure, then it is resolved at the kernel level itself, else it is passed to the user space. Here, the user can dynamically specify whether to allow or deny a system call if it is not already mentioned in the policy. Drawbacks of this tool are as follows:

1. It cannot work with the latest kernel.
2. It requires a training session.
3. It has some usability issues.

E. MBox

MBox introduces a file system layer between the host file system layer and the operating system [1]. The process running in the sandbox affects only the virtual file system layer and does not make any changes to the actual file system. Once the program has finished executing, the user can examine the virtual layer to see what changes have been made and decide which of these changes must be reflected on the host file system. The insertion of the virtual file system layer is done by allocating the sandbox a separate file on the host system. Any changes made by the running program are recorded here. This is possible because the sandbox rewrites the arguments of the system call to direct the effects of the system call to the required folder. In this tool when a sensitive program is called, it will point to a new page called read only memory. This memory is private and cannot be modified.

Feature	Janus	Bluebox	Mapbox	Systrace	Mbox
Security policy	Static	Multiple, Static	Policies for behavioural classes	Dynamic	No policy
System call interposition technique	proc	ptrace	proc	ptrace	ptrace
Disadvantage	No multithreading	Omits network system call	inflexible	Does not work with latest kernel	No security policy
Logging	No	No	No	No	Yes
Feature	Check system calls that manipulate file descriptors	List of harmless system calls	Rewrites arguments	Hybrid architecture	Virtual File System
Virtual file system	No	No	Yes	No	Yes

IV. NEW TECHNOLOGIES

Several new features have been introduced in the Linux Kernel recently. These features make it easier to confine running processes. Features which are of interest to our system are:

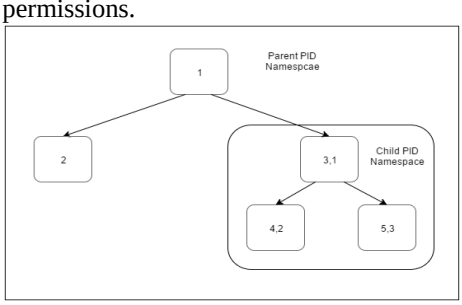
A. Seccomp-BPF

Secure computing mode (also called seccomp) is a facility that provides a mechanism for sandboxing in a Linux system. Version 2.6.12 of the Linux kernel which was released on 8th March, 2005, had seccomp merged into it. Seccomp puts a process into a "Secure" state wherein it allows the process to make only four system calls namely exit(), sigreturn(), read() and write(). The read and write only work with already open file descriptors. If the process attempts to make any system calls apart from these, the kernel will terminate it using SIGKILL. Hence we can say that seccomp does not essentially virtualize the resources of the system, but isolates them from the process entirely. An extension to seccomp, seccomp-bpf is implemented using Berkeley Packet Filter rules. It allows the use of configurable policies for filtering system calls. These filters can allow or deny specific system calls. Seccomp-bpf is available since Linux version 3.5. It has been used in software like Google Chrome/Chromium web browsers on Chrome OS and Linux, OpenSSH and vsftpd.

B. Linux Namespaces

Namespaces are an elegant way to confine processes. "**chroot**" in Linux is a way to change the apparent root directory for a process and its children. This way, the process can only access certain parts of the file system. Similarly, namespaces allow other aspects of the Operating System, like process tree, networking devices it can see, mount points, etc to be modified as well. Linux kernel maintains a single process tree. Given that a process has the appropriate permissions,

it can kill any other process. Process namespaces allow a process to view a completely isolated process tree so that it cannot kill the processes not created by itself even if it has the required permissions.



A network namespace allows processes to view a different set of networking interfaces. Even a different loopback interface.

Mount Namespaces allows processes to see a complete independent set of mount points. If the child process tries to change mount points, or the root directory, it will only be seen by the child process and will not be affected in the original mount point structure.

User namespaces allow a process to have root access within the namespace, without having access to processes not belonging in that namespace.

V. PROPOSED SYSTEM

A. *Seccomp-BPF*

It is much faster than ptrace alone. This enables us to run processes without causing significant overheads. The downside is that it is available on the recent kernels.

B. *ptrace*

ptrace is available on most linux systems. It allows the parent process to trace the child process by examining its registers, thus gaining access to the parameters it passes.

C. *GUI*

A casual user would prefer a graphical user interface which simplifies using the system. The user should feel at ease when defining policies, and enforcing restrictions.

D. *Namespaces*

For most basic purposes, it is essential to incorporate process namespaces and user namespaces. Process namespaces would help isolate two processes running in the system. User namespaces on the other hand, enables the child process to behave as the privileged user, but only within the namespace.

E. *Per app security policy*

This enables us to have more flexibility. We can choose restrictions for each individual apps. This will allow us to be more specific and impose restrictions separately for every new app as per our needs. This is much better than having a global security policy or different static policies for different classes of applications.

F. *Logging*

All the restrictions imposed on a program and the modifications made by it should be logged in a file so that it can be used for post-execution analysis. By evaluating the logged records of the processes, we can see which resources each application tried to access. From this we can judge which applications can be harmful by determining which applications tried to access resources that it need not have accessed.

G. *Virtual File system*

Much like what Mbox implements, it is beneficial to let a process have access to a virtual file system. All changes made to files can be shown to the user, and committed to the actual system only if the user wants to. This can also help us install software programs virtually and test them before installing to the actual system

VI. CONCLUSION

Review of the already existing sandboxing mechanisms showed us that there were a lot of drawbacks in each tool and that there was no ideal mechanism available to be used on a local Linux based system. In the system that we propose, we try to eliminate those drawbacks by maintaining optimal performance. Our tool not only supports unique security policies, a detailed logging mechanism and an interactive and user-friendly GUI, but also introduces a virtual file system layer. It allows the user to examine all the modifications caused by a running program and gives the user the choice to decide which of these changes must be committed. It reduces overheads without compromising on performance and also supports process isolation.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to our guide Prof. Laksmi Kurup for her guidance, encouragement and gracious support throughout the course of our work, for her expertise in the field that motivated us to work in this area and for her faith in us in every stage of this research.

We're grateful to Dr. Narendra Shekokar, Head of the Computer Engineering Department, for letting us use the department resources, labs and books.

We're highly indebted to our Principal, Dr. Hari Vasudevan for availing us with library books and relevant materials and also the SVKM management for providing us with popular and resourceful journals like IEEE and other online material that helped us.

REFERENCES

1. 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.
2. Graphical Security Sandbox For Linux Systems Topaktas, Cosay Gurkay (2014) Graphical Security Sandbox For Linux Systems. Masters thesis, National University of Ireland Maynooth.
3. Anurag Acharya and Mandar CoolRaje. 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.
4. David A. Wagner. Janus: An approach for confinement of untrusted applications. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1999
5. 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.
6. A Policy-driven, Host-Based Intrusion Detection System, ISOC Symposium on Network and Distributed System Security, San Diego, CA, USA, February 2002. ISOC