OS Course Project: Safebox
Software fault isolation using syscall prevention approach

Yuyang Rong
School of Information Science and Technology
ShanghaiTech University
Student ID: 69850764

Jianxiong Cai
School of Information Science and Technology
ShanghaiTech University
Student ID: 67771603

Abstract—**The grading server have to protect itself from being hacked by malicious program submitted from Internet, and protect test cases from being poached by untrusted program. To be general, a safe environment is needed for user to run untrusted program where user and his data can be safe.**

Currently, there are two ways to create such an environment. One is to use container or virtual machine to create a virtual environment to run the untrusted program, while the other is to track and limit the syscall made by untrusted program. There are pros and cons for both project.

In this project, we implemented a sandbox for safely running untrusted program, using the second way. Besides, we developed a rust library for trace syscall of another process.

1. Introduction

How to test a program, either for student homework or competition, is a tricky problem since the grading server have to run the program in order to grade the program under most situations. On the other hand, because the student homework or competition code is submitted from Internet, and is likely to be buggy and may damage the operating system running it. Besides that, some program may try to poke the test cases during grading and try to send it to the author.

Therefore, the student homework or competition code should be considered as untrusted program. The operating system need to protect itself from being damaged and prevent the program from poking test cases.

As the result, a safe environment is needed for user to run untrusted program where user and his data can be safe.

2. Motivation

2.1. Tread Model

As has mentioned before, the untrusted program may perform following operations, either intentionally or accidentally to damage the system or poke test cases from grading.

- Read files not owned by the current user
- Remove files owned (or can be accessed) by the current user, but should not be removed or changed by the program, like test cases
- Read in test cases and send it out through Internet.

As the result, there are a few requirements for the sandbox.

- Protect files not owned by the user
- Prevent program from removing files can be accessed by current user through file system permission checking but should not be removed, like test cases
- Limited Internet access for the program

2.2. Current Solutions

2.2.1. LXD. Currently there are many mal-ware isolation applications, one of the most famous is *lxd*. We dived into *lxd* and realized that it's designed based on the following principles:

If you can't see it, you can't destroy it. You can't tempering with things you can't see.

Good as this principle is, it causes unintended inconvenience. For example, Rust compiler have to check Github.com for extern crates if it is one of the dependences of the user program. But to make sure the data is safe, in *lxd*, network access is either entirely banned or allowed. Such situation can be really problematic.

2.3. Janus sandbox

Reasearchers at U.C. Berkeley create a safe environment for running helper application by monitoring every syscall it made. It follows this principle:

An application can do little harm if its access to the underlying operating system is appropriately restricted. [1]

The pros is that it used an approach which can individually limit access of each program. However, because it is a sandbox for helper application, it did not implement network access control at that time.

Thus we realized we need something better to help cope with the situation. We want network, file system, etc be available without damaging user's computer or server. That's how we started our project.

3. Project Contribution

- Implement Janus sandbox in Rust with a few modification
- Implement the sandbox to allow untrusted program only can connect to certain permitted IP address
- Implement a Rust lib to trace another process, based on ptrace with Rust FFI (Foreign Function Interface)

4. Design

4.1. Principles

So we came to our solution. As we have mentioned above, the *lxd* does not provide limited Internet access, instead, it either allow all Internet access or deny all. So we did not followed the principle of container / virtual machine to implement the sandbox.

Instead, we followed the principle of Janus sandbox, because the assumption Janus sandbox made, as mentioned above, is quite concrete.

In general, we will trace every syscall the untrusted program is issuing and decide if it's an allowance or a denial.

4.2. Difficulties

4.2.1. Enormous syscalls. Nice as tracing every syscall sounds, it's hard as soon as we realized the real scale the problem is.

The number of syscall are enormous, far beyond our expectation. For example, in x86_64 architecture, there are in total 332 syscalls.

Writing 332 different policies to deal with each syscall is is hard and laborious. Instead of doing that, because there are connections between some syscalls, like certain syscalls have to be performed before others, so there is a convenient way.

For example, each and every program have to *open()* a file before it can *read()* or *write()* it. Or, *socket()* have to be established before a network connection can be possible.

we decided to chock the untrusted program on those syscalls, to put pressure on where it hurts. For example, we allow *close()* , *read()* directly, but we carefully examine *open()* , whether the file is allowed, what it is opened for, etc.

In this way, we narrowed our work to much fewer syscalls like *open()* and *connect()*.

4.2.2. Killing a process. Not much program nowadays do not do *fork()* anymore, even students' homework use this syscall here or there. When a program *fork()* , tracing program do *fork()* too, and immediately begin to trace the child just forked by the untrusted program. After these finishes, we will start both parent and child.

The problem is, when a untrusted program *kill()* it's child, or one of it's children *kill()* another, the tracer can't tell the difference. Tracer have no way to know the *pid* about to be killed is another user program or one of one of tracee's children. Tracer program have to do inter communication to know what really happened.

Doing inter communication between tracers can be rather difficult can gains little, we then came up with a new solution that we change tracee's *uid*, thus every tracee's child will have the same *uid* and they can feel free to *kill()* each other, but once they try to *kill()* anyone else, operating system will stop them.

We tried to use *setuid()* to directly change uid, but soon we realized that it won't work since binary and outer folder doesn't belong to the new user and thus any operation will be denied by OS. The solution we have new is to use *lxd* to do it for us. *lxd* will change uid and thus do our job.

4.3. Alternative approaches

4.3.1. Limited file access. The first thing came into our mind that can be used for file access limitation is to write a configuration file. In this file we specifically state what file is allowed and what is not. This is easy to implement, but hard for users to use. After we run a few tests we realized that binaries usually need dynamic linked libraries and it's hard for user to list all of them.

Another way we considered is to directly "ask" file system to do it for us. Once we changed uid of the program, it can't access because file system will not allow it. However, in this way we may accidentally put too much constrain with get away. For example, Cargo have to access */home-/user/.cargo* upon running, such deed may cause yet another inconvenience. However, the get away is some untrusted program that need to run under *sudo*, thus file system can do nothing about it.

Finally we decided to use *lxd*(File system) and configuration file combined. *lxd* will limit the majority of tracee's invalid file access. For example, it prevents the tracee to access files of other user. If the file only exists on host system, the tracee can not name it in the lxd. If the file is in the container but owned by other user, the file system permission checking would deny invalid access.

Configuration file is a a list where files can be removed or changed. It helps the user when the grading process needs some file to be in the lxd container, like running a web server locally to test a web application. And the untrusted program have to be run as root. Under that circumstance, the configuration file would help the user a lot.

4.3.2. Limited network access. If the program to be grade is a network program which does not necessary relay on the Internet, like a web browser. One alternative way is:

1) Disable the Internet access like using a lxd container without network configuration.
2) Test the program through *loopback network interface* (usually *127.0.0.1*)

However, this approach for program requiring network access but not Internet access. If the program need Internet access to run, like a Rust program may have dependence on

Rust Cargo, which is online, the grading system can not do its grading job in the approach.

As the result, in our design, the sandbox give limited network access to the untrusted program by monitoring the destination ip address of every network connection.

5. Implementation

5.1. ptrace

Linux provides a function:

ptrace(request, pid, addr, data)

This function can be called both by untrusted program or tracing program. In user manual, the program being traced is called *Tracee*, we will call it tracee from now on.

What ptrace does is that it stops syscall and have the privilege to examine memory, registers, etc. the tracee is using. Every time the tracer calls this function, it will be put to wait. It will wake up on two conditions: either tracee issues an syscall then OS wakes tracer up and continue doing the syscall until tracer says so by a *ptrace(CONT, pid, void, void)* call or OS finished a syscall it wakes tracer up before it hands control to tracee.

5.2. Choice of architecture

ptrace may have the ability to alter registers, it don't necessarily have the ability to know the syscall the tracee is making. Different architectures store syscall number and syscall return value in different registers. For example in MIPS they are stored in *$a0* x86 *%eax*, x86_64 *%rax*.

MISP is not widely used in PC, it's mostly used in embedded processors and they don't need that much software isolation. Besides, it rather an old architecture.

We considered making a sandbox on x86, but it's an old architecture too. We totally give it up because we realized that x86_64 can run x86 program too, so if we use x86_64 architecture, only slight changes shall be made then sandbox can run on x86.

We decided we will use x86_64 architecture since this is the most widely used architecture, easier for us to program when both of us are using x86_64 architecture laptop.

5.3. Programming language

ptrace() is a C function, so we thought let's do it in C. But after we carefully examined several programming languages we decided to use Rust.

5.3.1. C. C is the first programming language came into our mind. But after we toyed a little with C and *ptrace()* we realized that *ptrace()* is rather a complicated function, we want to encapsulate it to make our life easier. However, C is not OOP, so we abandoned it and started to consider C++.

5.3.2. C++. C++ is good at encapsulation, but too week at type system. We build this sandbox to be safe, not to introduce more volubilities. So instead of C++ we started looking Rust.

5.3.3. Rust. Rust have it's pros and cons too. There is not *ptrace()* implemented in Rust, we can only use this function through extern crate *libc*, so we are forced to add a lot of *unsafe* to our code. However, life time system and ownership system allow us to code more confidently.

5.4. Encapsulation

We found a finished ptrace library on crates.io, but it couldn't compile because of certain dependencies are missing. We tried to fix it, but we failed because the lacking dependencies have been removed by Rust team.

So we decide to write ourselves a library to do that job. We created a structure contains a pid and other necessary components listed below:

```
pub struct Tracee {
  pid: libc::pid_t,
  allow_all: bool,
  entry_flag : bool,
  last_syscall: u64,
  ip_connected: Vec<String>,
  file_opened: Vec<String>,
}
```

allow_all is a sign in case the user don't want any protection. *entry_flag* shows if the tracee is exiting a syscall or entering. The last two vectors are just logs used to tell tracee's behavior.

We can initialize a Tracee by telling the command so that the constructor will call *fork()* first and the child will call *ptrace(TRACEME, pid, void, void)*, thus a program is set up and ready to go.(The program won't start until tracer says so.)

We now provide interface for certain widely used functions, like *take_regs()* and *do_continue()*. These functions do not require any arguments but pid, using *ptrace()* to do that can be laborious, so we finished the interface.

5.5. The framework

5.5.1. Reading configuration file. Currently Safebox have 2 configuration files, the IP address and file allowed to access.

When the sandbox is launched, it would read in the two configuration file and stores it in vectors. After that, each time the untrusted program perform file access or network connection (not in *allow all* mode), it would check if the file location / target ip address is permitted.

5.5.2. Initialize the tracee. Good thing is this have been done in our own library, all we need to do is to take command from command line and put them into a vector, give that to constructor.

After construction, the tracee is halted at the first *execvp()*, user have to manually start the tracee by calling *tracee.do_continue()*

5.5.3. Tracing according to syscall. The main program goes into a loop which will not break until the tracee is stopped because it exited instead of an issue of syscall. In the loop we will intercept every syscall, each twice.(In and out) Now we don't do any examine on return value of a syscall so we directly allow it by saying *tracee.do_continue)()*. However, if the tracee is entering a syscall, that's another story. We will make decisions depend on the syscall the tracee is making.

5.5.4. Allow or Deny. Currently we focus on two syscalls, *open()* and *connect()*

In terms of *open*, we only allow files inside current working directory, outside /home or specifically permitted in configuration file. We believe if tracee is toying files inside current working directory, little harm can be made. If tracee tries to tank files outside /home, operating system will prevent it.

5.6. Network

There are different ways to limit the access of certain program to network. *lxd*, the Linux Containers, limit the network access by configuring the whole container, all programs in one container will either have network access or not. Taking advantage of syscall-level sandbox, every process is separate, thus monitoring one particular program's network connection is possible. Besides, by implementing in this way, the sandbox can allow the untrusted program only be able to connect to certain IP address.

There are four syscall a client program would use to make a network connection.

1) *socket()*
   get a new socket.
2) *connect()*
   connect the socket to a remote IP.
3) *recvfrom() / sendto()*
   communicating with the remote IP though the socket.

Because the connect syscall is the only syscall whose arguments contain the IP address, tracing this syscall can get the IP address, then the sandbox can decide whether this connection is allowed or not.

For example, if the untrusted program is only allowed to connect to Github.com, when the sandbox found the IP address which untrusted program is trying to connect to is beyond the white list, this syscall will be rejected.

6. Rust Experience

After finish this project, we now have a better feeling about Rust programming language. It's pros and cons we will list our feelings here.

6.1. Pros

6.1.1. Trait system. It is really powerful, it allows us to add additional method to predefined types. When doing file path parsing, a new method need to be add to String type, Trait system allows it.

6.1.2. Mutability system. In C++, variables are assumed mutable. Thus it is dangerous to return a pointer to a variable in a struct, because the user may accidentally change the variable and could easily corrupt the struct. Taking advantage of Rust, all variables are assumed immutable, thus the user can't change the variable. In this way, struct can't be corrupted by buggy program.

6.2. Cons

Although Rust is said to be a system programming language, it still lack certain "system" interface. We have to heavily rely on *libc* to do the programming and have tons of *unsafe*s in it, which is not that desirable.

7. Limitations & Future Work

Thanks to Prof. Chen's advice, we tried to use user system in Linux to protect user's data. If the file opened doesn't belong to tracee, the system will deny it and return -1 on *open()* call. Now that we are hiding out safe-box inside *lxd*, we would like to look for ways to create a temp user ourself and provide protection without the help of *lxd*.

7.1. Network

Currently, the sandbox only support untrusted program making connect as a client. If the untrusted program is a server application, the syscall it would use would use would be socket, bind, listen, and accept.

In the syscall "accept", the address of client on the other end of the connection would be provided, so it is reasonable to monitor this syscall to decide whether the connection should be allowed or not.

8. Availability

Please check Github.com for our Safebox.

Acknowledgment

We would like to thank Prof. Chen for his brilliant course and his guidance for our project. He provided us with such a great opportunity to do system programming and rust programming ourselves.

References

[1] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 5th USENIX Security Symposium*, 1996.