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*—How to test program, either for student homework or competition, is a tricky problem since the grading server have to protect itself from being hacked by malicious program submitted from Internet, and to 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. We propose that we track every syscall the untrusted program issues, thus we can trace it's movement and status and make sure it remain trustworthy.**

1. Motivation

1.1. Real life problem

From grading system's point of view, student's homework or code need to be tested should be considered untrustworthy. However, in order to do the grading job, the server have to run all students programs.

Currently there are many malware isolation applications, one of the most famous being 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.

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.

1.2. Principles

So we came to our solution. But first we have a basic assumption:

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

After we read the reference paper we find this assumption ideal and solid. So based on that we derived the following philosophy:

If you have to ask for my permission for every syscall you make, you can't damage things because I will not allow it.

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

1.3. Difficulties

1.3.1. Outnumbered. 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 328 syscalls.

Writing 328 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.

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.

So 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()*.

1.3.2. Killing a children. 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.

2. Implementation

2.1. 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.

2.2. Architecture & System choice

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.

2.3. 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.

2.3.1. 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++.

2.3.2. 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.

2.3.3. 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.

2.4. 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.

2.5. The framework

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

2.5.2. 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()*

2.5.3. 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.

2.5.4. 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.

2.6. 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) receive from / send to
   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.

3. 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.

3.1. Pros

3.1.1. Yuyang Rong. I really like Trait system, it allows me to add additional method to predefined types. When doing file path parsing, I really need a new method added to String type, Trait system allows it.

3.2. Cons

3.2.1. Yuyang Rong. 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.

4. Limitations & Future Work

4.1. 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.

5. Conclusion

The conclusion goes here.

6. 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.