Building an in-kernel, per-process sandbox

Vinod Ganapathy (vg@iisc.ac.in)

E0-256 (Autumn 2024), CSA, IISc Bangalore

This document describes the project that you will do as part of this course. You are required to do the project alone (no teams). The goal of this project is to implement an in-kernel, per-process sandbox. That is, we would like to build a kernel module that accepts a per-process policy regarding what calls (library calls in this case) the process can execute (and at what points during its lifetime it can execute those calls), and enforce that policy within the kernel.

The importance of such sandboxes and how they help sandbox hijacked processes has been documented in several research papers. As a first step to understand the background for the project and the problem statement we would like to address, read the following papers:

- "A secure environment for un-trusted helper applications (confining the wily hacker)" by Goldberg et al. from USENIX Security 1996 [GWTB96],
- "A sense of self for UNIX processes," by Forrest et al. from IEEE Symposium on Security and Privacy 1996 [FHSL96],
- "Efficient context-sensitive intrusion detection," by Giffin et al. from the Network and Distributed Systems Security Symposium 2004 [GJM04].

These papers are by no means a comprehensive list of papers on this topic. Moreover, these are all old papers, built on old systems, but it should give you a good idea of some of the concepts. We will build on the ideas expounded in these papers. Note that these papers describe *system call* sandboxes. For several reasons guided by experience in prior years, we will relax this somewhat and have a library call sandbox (particularly libc sandbox).

In particular, in this project you will build a sandbox that enforces a statically-extracted policy of acceptable library calls.[1] The core mechanism of this sandbox that enforces the policy will be implemented within the Linux kernel.[2]

The project will consist of several parts to implement the following functionality. You will be given a source level C program. You will be building tools to:

1. Analyze the source level C program using a compiler, and emit a policy of acceptable library calls generated by that C program;
2. Build a Linux kernel-level enforcement engine to enforce this policy. You are given the choice of building a kernel module, using the seccomp or eBPF framework to enforce it on the process to be protected. You will be working with simple single-threaded programs and accomplish tasks (1) and (2).

The rest of this document will describe each of these tasks in detail. You will have significant flexibility in several aspects of the project. I will provide pointers, but you are free to use any tool of your choice.

Important Note: Please also note that there will not be much hand-holding in this project (neither from me nor your TAs). You will be expected to do a significant amount of work on your own, including figuring out what tools you'd like to use, how things work, the design decisions you take, and figuring out how best to implement your proposed design. In this effort, you are free to look up forums on the Web, read papers, read code/comments, and also freely discuss ideas with your classmates (with suitable credit provided to your classmates), just the way a software developer works in industry. However, *all the implementation must be your own* and any evidence of cheating in this aspect will

[1] In contrast to the work of, say, Forrest et al. [FHSL96], which extracted acceptable call traces using dynamic analysis.

[2] In contrast to the work of, say, Giffin et al. [GJM04] that implemented it using binary rewriting on SPARC binaries.

be dealt with very strictly. Again, this just mimics what happens in the software industry—software developers in industry that plagiarize or claim credit for work taken from others generally have their employment terminated. This is all an effort to get you to *become independent software developers and system builders* and *learn how to do research*. We will only be interested in the final outcome of each part, and you will have to figure out all the details on your own based on the description in this document.

Each part requires you to pick up a different skillset (program analysis and rewriting using a compiler, linux kernel development). Be warned that *this is a significant implementation effort*. You will be expected to work regularly — working irregularly or in a bursty fashion will not serve you well — and we will not be providing any extensions to the due dates stated in this document. Please consider these aspects carefully, especially in relation to other courses you may be taking during the semester.

1 Part 1: Extracting a library call policy from a C source program and emitting a binary with "dummy" system calls

Due Date:	October 10, 2024.
Learning:	How to work with a compiler to extract control-flow graphs and learning how to instrument emitted code with dummy system calls.
Weightage:	50% of the overall project score.

You are given an application that may have several C source files. The first step of the project is to extract a *library call policy* for the given program.

A library call policy is a set of rules that determines what library calls (from a given library, say the standard C library, aka libc) can be issued when at different stages during the execution of the program. Prior systems have demonstrated expressive and powerful call policies, in the form of automata [GJM04], together with the set of permitted arguments for each call.

For the purposes of this course, the library call policy will be, for each procedure in application, *the sequence of library calls* that that procedure can legally invoke. Consider this code snippet, from a procedure `foo`, for instance"

```
foo(...) {
    FILE *fp;
    fp = open ("/home/alice/project.txt", "r")
    read (fp, ...)
    close (fp);
}
```

Note that these calls to `open`, `read`, and `close` are calls to the standard C library. The library call policy for `foo` is the sequence of calls:

`open`→`read`→ `close`.

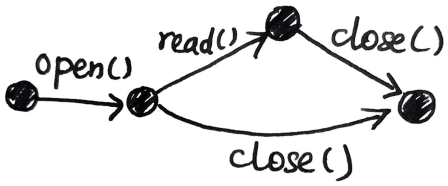
If, at runtime, you observe that a library call is being invoked that does not match this sequence, you can immediately raise an alarm saying that an exploit is likely underway. Of course, code does not always look this simple, and in the presence of conditionals and loops, you will need a more expressive representation of the call policy than mere sequences. We will go in for a *library call graph* representation of the policy, that implicitly denotes the sequence of library calls that can be legally invoked by the procedure.

The *library call graph* depicts the control-flow graph of the program projected down just to library calls. To be precise, the library call graph of a program consists of nodes and edges where (a) nodes denoting abstract states of the program, and (b) edges are labeled with library calls, showing transitions between these abstract states. During any execution of the program, you can determine the library calls that the program will issue by just following one path along the library call graph. Thus, the library call graph determines the library call flow policy of this program, and at runtime, you must expect to see the program issue library calls exactly in accordance with this flow graph.

For example, consider the program shown below.

```
foo(...) {
    FILE *fp;
    fp = open ("/home/alice/project.txt", "r")
    if (fp != NULL) {
        read (fp, ...)
    }
    close (fp);
}
```

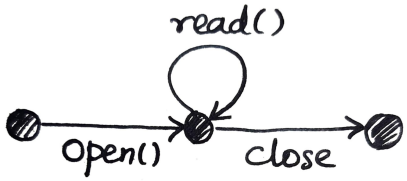
The library call graph for this program will look like so:



Observe that by storing it as a graph, your library call flow graph compactly represents both the permissible library call sequences in `foo`. Every path from the start node of this library call flow graph to the end node denotes a permissible library call sequence.

The graph will look more interesting in the presence of loops in the program:

```
foo(...) {
    FILE *fp;
    fp = open ("/home/alice/project.txt", "r")
    while (...) {
        read (fp, ...)
    }
    close (fp);
}
```



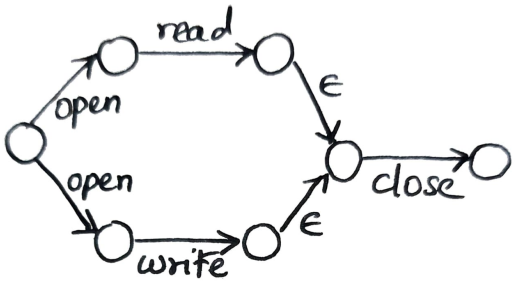
Note that this library call graph (which is in fact a finite automaton), denotes the set of sequences denoted by this regular expression: `open`.(`read`)*.`close`.

Now, it is true that the number of `read` calls can be bounded if you are aware of how many times the `while` loop executes, and that the library call policy above is overly permissive in the sense that it may allow library call sequences that do not occur during any real execution of the program. However, expressing such constraints on the number of times the loop executes is undecidable in general (equivalent to the halting problem of Turing machines), and so, we will stick to the rather simple policy shown above.

The example thus far has shown a library call flow-graph that represents a deterministic finite automaton. But in the most general case, the automaton can be non-deterministic, and the library call policy that you extract will in fact correspond to a non-deterministic finite automaton. Here is a simple example that shows how non-determinism arises.

```
foo(...) {
    FILE *fp;
    if (...)
```

```
        fp = open ("/home/alice/project.txt", "r")
        read (fp, ...)
    }
    else {
        fp = open ("/home/bob/project.txt", "w")
        write (fp, ...)
    }
    close (fp);
}
```

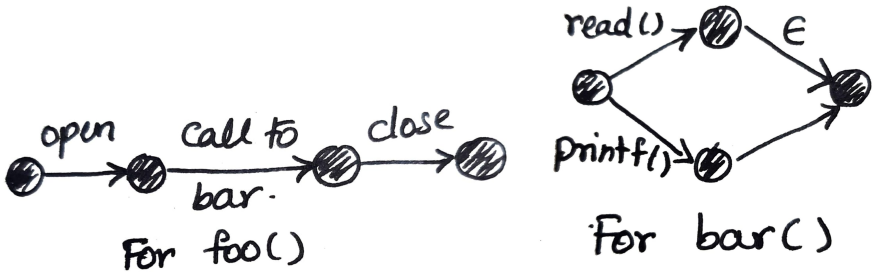


Observe that when you first see the `open` call, you do not know whether the program will subsequently issue a `read` or a `write` call. Thus, the resulting library call flow graph is can be non-deterministic Note the ϵ edges that appear in a non-deterministic finite automaton also appear in this graph. (yes, I know there are ways in which you can make this particular graph a deterministic automaton; I just wanted to illustrate how non-determinism may arise).

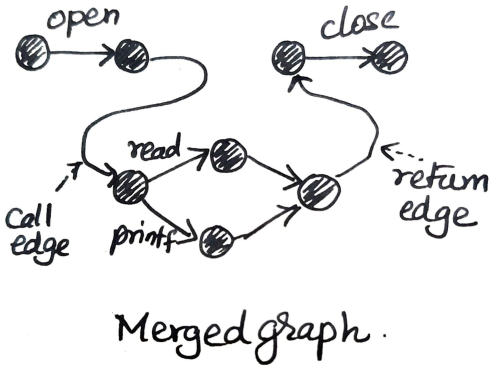
Function calls and returns add a further element of complexity. Consider the code example below.

```
foo(...) {
    FILE *fp;
    fp = open ("/home/alice/project.txt");
    bar (fp);
    close (fp);
}

bar (FILE *fp) {
    if (fp != NULL) {
        read (fp ...);
    }
    else {
        printf ("could not read file);
    };
}
```



Note that there are two functions, each of which has its own library call policy. You will need to extract library call graphs on a per-procedure basis. Once you have done so, you must have a way to "paste" the library call policy of each function at every place it is called, to obtain a complete library call graph for the full program. Here, for example, is the complete library call graph of the program.



A number of practical complications (related to non-determinism) arise when you try to create a policy like this for the whole program. You are not expected to implement solutions for all these complications, but you are expected to be aware of them. The paper by Giffin et al. provides an excellent overview of the challenges that arise [GJM04].

You are free to determine the format in which you want to store the library call graph once you extract it. However, an important practical problem remains once you have extracted the library call policy: *how are you going to enforce it?* To enforce that the policy is followed by a running application, in the second part of the project, we will be building a kernel-level sandbox. However, there is an important practical problem – namely, that library calls are not really visible to the kernel – the kernel only sees system calls. How is one to enforce the library call policy within the kernel if the only API visible to it is that of system calls?

To solve this problem, we will use a rather classic trick. We will modify the binary code emitted for the C program so that before each library call, we will place a dummy system call (let us call this system call `dummy`). This `dummy` system call will take a single integer argument that unambigiously identifies the library call. Inserting `dummy` system calls has the effect of modifying the program previously discussed to the following program instead:

```
foo(...) {
    FILE *fp;
    dummy(1);
    fp = open ("/home/alice/project.txt");
    bar (fp);
    dummy(2);
    close (fp);
}

bar (FILE *fp) {
    if (fp != NULL) {
        dummy(3);
        read (fp ...);
    }
    else {
        dummy(4);
        printf ("could not read file);
    };
}
// Assume that 1 is the identifier that we use to denote the open library call
```

```
// in the argument to the dummy system call, 2 is the identifier for close,
// 3 is the identifier for read and 4 is the identifier for printf.
```

Although we have shown the `dummy` system call inserted in the C source program, you do not have to insert it in the C program – you can automatically do this in the compiler during the code generation process.

For this part of the project, we strongly recommend using the LLVM compiler toolchain. Use the LLVM compiler toolchain to produce the in-memory LLVM Intemediate Representation (LLVM IR) representation of the program, and write an LLVM pass to automatically analyze the program and extract a library call policy for it (i.e., an automaton describing the library call policy). We will need a method to visualize the library call graphs to make sense of them. Please output your library call graphs in a format (or also provide scripts that convert the graphs into a format) that is compatible with the `graphviz` software [Gra] (the input language for `graphviz` is called DOT, and is a very simple, logical way to represent graphs, so we highly recommend using it). We will be using `graphviz` to visualize your library call graphs during grading, therefore providing compatibility with graphviz is key.

Subsequently, write an LLVM pass to automatically insert `dummy` system calls (with suitable identifiers determined for library calls) into the LLVM IR. The binary program that will result from this LLVM IR will contain these `dummy` system calls. This program will not automatically run on a given Linux system (because you haven't yet implemented this system call in the kernel), but we will do that in the second part of the project.

Deliverables. Please submit the following:

- Your LLVM-based tool that takes as input a C program and outputs a library call graph for libc in `graphviz`-compatible format. This LLVM-based tool must also ensure that the LLVM IR includes `dummy` system calls ahead of each library call in the program
- A detailed README on how to use your tool.
- Please note, your code must work on the mbed-tls benchmarks (see Section 3)

While I describe the deliverables in terms of LLVM, you are free to use alternative tools of your choice. For example, you can use `gcc` or any other compiler if you are more comfortable using them. Also, you do not have to insert `dummy` calls using a compiler pass – you can do them using a C source-to-source transformation tool (if you are aware of and comfortable using any such tools already) to insert the `dummy` system calls.

2 Part 2: In-kernel enforcement of the library call graph

Due Date:	November 20, 2024.
Learning:	Learning how to build a kernel-level sandbox, and (as learning outcomes on the side), learning how to build and compile the Linux kernel and work with it in a virtual environment.
Weightage:	50% of the overall project score.

For this part of the project, you will build an in-kernel monitor that accepts a binary program containing `dummy` system calls, its library call graph (essentially the program's security profile, where each `dummy` system call serves as a proxy for the corresponding library call), and then enforce that policy on on the running process of that binary program. You are free to use any tool of your choice for this project, such as seccomp, eBPF or writing your own kernel module if you wish. However, I will *not* permit user-space frameworks (e.g., those based on ptrace) to enforce this library call policy.

For this part of the project, please use any of the newer versions of the Linux kernel (Linux 6.0.x would be ideal). Use a virtual machine (such as Virtual Box) or system emulator (such as QEMU) to run the modified kernel. We *do not* recommend doing your Linux kernel development and compilation within the virtual environment, as it will be exceedingly slow. It is best to develop the source code on your local laptop or desktop, compile the kernel on that machine, and then just run the booted kernel image within the virtual environment. Of course, the program that you will sandbox will have to run inside this virtual environment, so you can copy it there (and also copy the library/dummy system call graph that you extract) and simply use the virtual environment to test whether your sandboxing works.

Your first task will be to modify the kernel to have a new `dummy` system call. This system call will have the API:

```
void dummy (int libcallno);
```

That is, it returns no values, and takes a single integer argument denoting the ID of the library call ahead of which it is to be placed. Each unique library call will have a unique ID, and the interpretation of this ID is entirely up to you (i.e., you are free to assign IDs based on the extracted API of your library). The system call also does not have to do anything functionally, except to have hooks to invoke your kernel monitor – the precise nature of the hook depends on what you use to implement your kernel monitor, be it eBPF, seccomp, or your own kernel module. If you wish, you can implement the functionality of your security monitor (described below) in the body of the `dummy` system call itself, that is entirely your choice.

Your kernel monitor must enforce policies on a per-process basis. When you start the process (e.g., using the full path name of the executable of the process), the enforcement engine that you write must load the corresponding security profile for that process. The security profile will be the dummy system call graph (i.e., library call graph) that you extracted in Part-1 of the project. The kernel can store such security profiles in the file system, and index the security profile using, say, the full path name of the executable. That way, when you start the executable (e.g., via clone or fork), the corresponding security profile is loaded.

The kernel level enforcement module that you build must load the automaton from the first part, and advance the frontier of the automaton based on the dummy system call issued by the process. When running a process, the kernel enforcement engine's main functionality will be to advance the automaton's frontier in accordance with that automaton. Thus, for example, if you decided to have a non-deterministic finite automaton policy, and you see a dummy system call corresponding to, say, `read`, then you must advance the frontier pointer and advance it across an automaton transition edge labeled `read` emanating from that state in the frontier. If no such transition exists, then the frontier state is killed (as is standard in all NFA operation). If all frontier states of the NFA are killed (at any point in time), then that means that the library call policy is violated, and you have discovered an attack.

3 Benchmarks

One of the most important aspects of any systems research is to evaluate the system that you build (its effectiveness, performance, etc.) against a set of benchmarks. We will have the same expectation in this course. We will evaluate your code submissions by running them against `mbed-tls` [mbe], a popular TLS library. `mbed-tls` can be compiled in a variety of different ways (with various modules included or excluded), and you will have to tell us in your submission how you compiled `mbed-tls` during your testing.

First ensure that you can run the compiled binaries, and measure their performance on your system. You will be extracting the system call graph from these binaries, and will have to sandbox these binaries with your in-kernel enforcement mechanism.

References

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for UNIX processes. In *IEEE Symposium on Security and Privacy*, 1996. `https://doi.ieeecomputersociety.org/10.1109/SECPRI.1996.502675`.

[GJM04] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium*, 2004. `https://research.cs.wisc.edu/wisa/papers/ndss04/dyck.pdf`.

[Gra] The GraphViz graph visualization framework. `https://graphviz.org/`.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *USENIX Security Symposium*, 1996. `https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/goldberg/goldberg.pdf`.

[mbe] https://github.com/Mbed-TLS/mbedtls.

Note: Some of the URLs above point to papers that sit behind a paywall. Access them from a machine in the IISc network (or connect via VPN), and you will have access to the paper from IISc's network.