Building an in-kernel, per-process, system-call sandbox

Vinod Ganapathy (vg@iisc.ac.in)

E0-256 (Autumn 2022), 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, system-call sandbox. That is, we would like to build a kernel module that accepts a per-process policy regarding what system calls the process can execute (and at what points during its lifetime it can execute those system calls), and enforce that policy within the kernel. Why is such a sandbox interesting? The basic motivation is that to do anything interesting, an exploited program must avail of the services of the kernel. And the mechanism to do so is a system call.

The importance of such system call 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.

In particular, in this project you will build a system call sandbox that enforces a statically-extracted system call policy.[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 binary program (in x86/64 assembly). You will be building tools to:

1. Extract the system call policy from the binary program using static analysis of x86/64 binaries;
2. Build a Linux kernel module using the Linux Security Modules (LSM) framework to accept this system call policy and enforce it on the process to be protected. You will be working with simple single-threaded, statically-linked programs and accomplish tasks (1) and (2).
3. [Bonus] Once these steps are complete, you will extend the kernel module and the policy extraction framework to handle dynamically linked libraries.

The rest of this document will describe each of these tasks in detail. You will have significant flexibility in several aspects of the project, e.g., in deciding which binary analysis tools you will use and so on. 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

[1] In contrast to the work of, say, Forrest et al. [FHSL96], which extracted acceptable system 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.

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 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, linux kernel development, learning about dynamic linking, and multi-threading). 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 system call policy from a x86/64 binary program

Due Date:	September 27, 2022.
Learning:	How to write a binary analysis tool (for x86/64) and extract a flow graph from a given binary.
Weightage:	50% of the overall project score.

You are given a statically-linked, single-threaded binary program in x86/64 assembly code. You can assume that the binary has debugging information present. The first step of the project is to extract a *system call policy* from the given binary program.

A system call policy is a set of rules that determines what system calls can be issued when at different stages during the execution of the program. Prior systems have demonstrated expressive and powerful system call policies, in the form of automata extracted from the program binary [GJM04], together with the set of permitted arguments for each system call.

For the purposes of this course, the system call policy will be, for each procedure in the binary, *the sequence of system 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);
}
```

The system call policy for `foo` is the sequence of system calls:

`open`→`read`→ `close`.

Please note that we have shown the example in source code for simplicity, but you will be expected to work with x86/64 binary programs as input.

If, at runtime, you observe that a system 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 system call policy than mere sequences. We will go in for a *system call graph* representation of the policy, that implicitly denotes the sequence of system calls that can be legally invoked by the procedure.

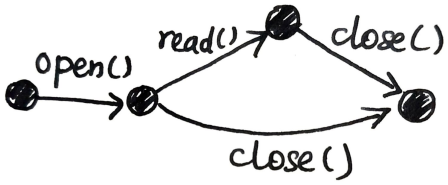
The *system call graph* depicts the control-flow graph of the program projected down just to system calls. To be precise, the system call graph of a program consists of nodes and edges where (a) nodes denoting abstract states of the program, and (b) edges are labeled with system calls, showing transitions between these abstract states. During any

execution of the program, you can determine the system calls that the program will issue by just following one path along the system call graph. Thus, the system call graph determines the system call flow policy of this program, and at runtime, the kernel must expect to see the program issue system 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);
}
```

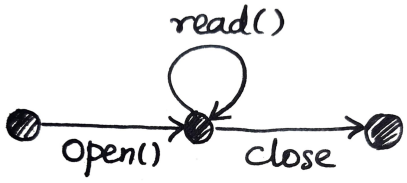
The system call graph for this program will look like so:



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

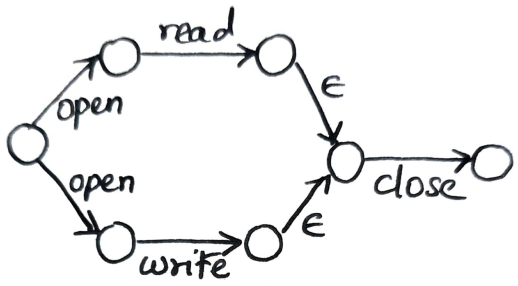


Note that this system call graph (which is in fact a finite automaton), denotes the set of sequences denoted by this regular expression: `open`.(`read`)*.`close`.

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 system call policy above is overly permissive in the sense that it may allow system 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 system call flow-graph that represents a deterministic finite automaton. But in the most general case, the automaton can be non-deterministic, and the system 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);
}
```

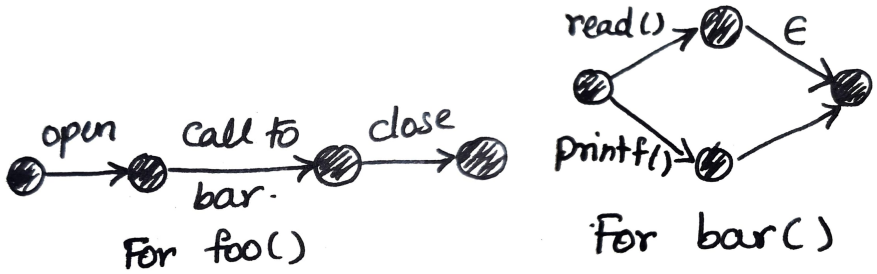


Observe that when you first see the `open` system call, you do not know whether the program will subsequently issue a `read` or a `write` system call. Thus, the resulting system call flow graph is can be non-deterministic Note the ϵ edges that appear in a non-deterministic finite automaton also appear 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.

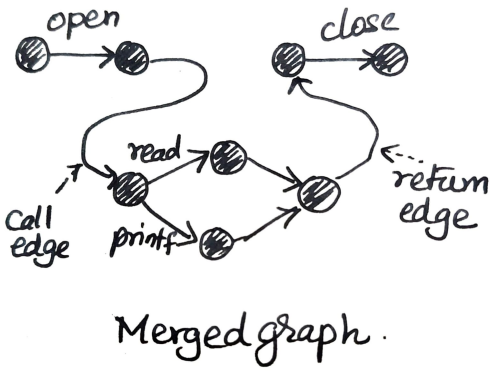
```
foo(...) {
    FILE *fp;
    fp = open ("/home/alice/project.txt");
    bar (fp);
    close (fp);
}

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 system call policy (well, `printf` is not *really* a system call, but I'm only illustrating a concept here).

You will need to extract system calls on a per-procedure basis. Once you have done so, you must have a way to "paste" the system call policy of each function at every place it is called, to obtain a complete system call graph for the full program. Here, for example, is the complete system 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 system call graph once you extract it (after all, you will be parsing it within the kernel, so you can decide what format to store it and parse it). It is best to annotate each edge of the system call graph with the system call along that edge, along with the address in the program where this system call is issued. Also decide on your vocabulary of "system calls," since that term is rather loaded. You are free to use the term system calls in the purest sense, i.e., the set of 300+ system calls supported in Linux (`https://man7.org/linux/man-pages/man2/syscalls.2.html`), or go in for the larger set of "system calls" (really wrappers) as implemented by libc. If you go in for the latter approach, for example, you can consider `printf` as part of your "system call" vocabulary.

For this part of the project, we strongly recommend using the binary analysis tool `angr` [Ang, SWS+16] to extract the system call graph. We will need a method to visualize the system call graphs to make sense of them. Please output your system 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 system call graphs during grading, therefore providing compatibility with graphviz is key.

Pro tip: Static linking. To test whether your methods work, you will need to create statically-linked benchmark programs. As mentioned earlier, we will be working with statically linked programs (i.e., all libraries used by the program are statically linked to the binary even before it starts execution). How does one create a statically linked binary? Well, first you've to know what libraries your program uses. And then you've to statically link those libraries

against your program binary. There are plenty of resources available on the web that teach you how to statically link your program against the libraries it uses. Feel free to consult those resources to produce a statically linked binary.

Deliverables. Please submit the following:

- Your angr-compatible x86/64 static binary analysis tool that takes as input a statically-linked x86/64 executable and outputs a system call graph either in `graphviz`-compatible format, or your custom format.
- If you output your graphs in custom format, then a script that converts the system call graph into `graphviz` format.
- A detailed README on how to use your tool.
- Please note, your code must work on the Embench-IOT benchmarks (see Section 4)

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

Due Date:	November 20, 2022.
Learning:	Learning how LSM works, building an LSM-based kernel module that enforces system call policies, 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 a kernel module (using the Linux Security Modules, or LSM, framework) that accepts a binary program, its system call graph (essentially the program's security profile), and then enforce that policy on on the running process of that binary program.

The LSM famework is a set of "hooks" or callbacks that are inserted into the Linux kernel. These hooks are inserted at various points in the kernel where a system call is about to interact with a critical kernels data structure. You must read the original LSM paper from USENIX Security 2002 [WCM+02] to get an idea of the designers' intentions behind creating LSM, but keep in mind that the LSM implementation in the modern Linux kernel has diverged significantly from the implementation described in the original paper. LSM allows you to implement a security module and then register suitable functions of the security module with these hooks. Every time the hook is encountered, your function will be invoked (via function pointers). Various sophisticated and widely-deployed security modules (such as security-enhanced Linux, AppArmour and so on) have been implemented using the LSM framework.

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 system call graph that you extract) and simply use the virtual environment to test whether your sandboxing works.

You can see several example LSM modules in the Linux kernel source code in the security subsystem: `https://elixir.bootlin.com/linux/latest/source/security`. Learn how to build kernel modules by looking at these examples, or sources like this: `https://www.youtube.com/watch?v=Y0QZpan5LbU`.

You will need to build your LSM module in C, compile it into a loadable kernel module, and then insert it into the kernel using the `insmod` command. This will ensure that your security module does not have to be integrated into the kernel build itself, and can be loaded and unloaded on-demand as a kernel module.

Your kernel module 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 LSM kernel module that you write must load the corresponding security profile for that process. The security profile will be the system 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 LSM module's functionality for clone or fork will be responsible for loading the security profile.

The LSM kernel module that you build must load the automaton from the first part, and advance the frontier of the automaton based on the system call issued by the process. Thus, your LSM module must implement a hook for every system call, and the only functionality of this hook 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 system call policy, and you see a system call for, 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 system call policy is violated, and you have discovered an attack.

3 [Optional Challenge] Part 3: Accommodating dynamically-linked libraries

Due Date:	November 23, 2022.
Learning:	Learning how dynamic linking works, and learning how to patch the system call policy dynamically when dynamic linking happens.
Weightage:	Successful implementation of this kernel module will add 5 bonus points to your **overall score in the course** out of 100.

Please contact me in person if you are interested in pursuing this extra credit assignment and I will provide the details.

4 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 single-threaded benchmarks in **the Embench-IOT benchmark suite** [Emb]. This is a set of freely-available C programs, on which you should ensure that your code works. Note that although the source code is available for these benchmarks, we expect your methods to work on *binaries*. Thus, download these benchmarks, compile them, and work with the resulting binaries. Ensure to statically link these binaries with the corresponding libraries that they use (if you attempt the bonus challenge—Part 3, you can relax this assumption).

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

[Ang] The angr binary analysis framework. `https://angr.io/`.

[Emb] The Embench-IOT benchmark suite. `https://github.com/embench/embench-iot`.

[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.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[WCM+02] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002. https://www.usenix.org/legacy/event/sec02/wright.html.

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.