CS261 Scribe Notes: Software Security 1 - Sandboxing

Scribe: Cameron Rasmussen

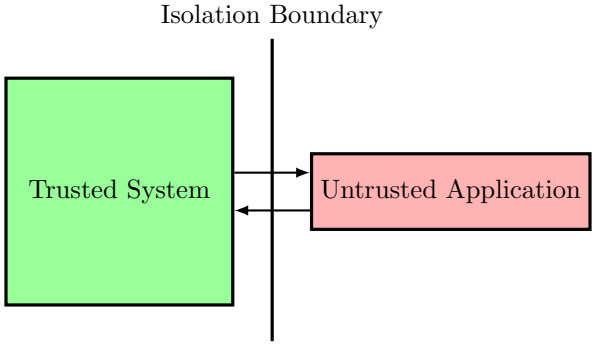
October 24, 2018

1 Introduction

It is often the case that code is being run locally on our system that cannot be completely trusted, a prime example being web applications. In the web context, simply by visiting a new website, our system will begin running new untrusted code. The problem is that simply preventing the code from running is not a solution from the usability standpoint because we visit new websites everyday. Furthermore websites commonly use a very large toolchain that can incorporate a large set of imported libraries or connected services, so whitelisting what domains are allowed to run on our system can become an exercise in frustration and still is not perfect without exhaustive auditing of the code being run. The common solution to this problem is sandboxing.

2 Sandboxing

The key idea for sandboxing is to isolate resources and/or the environment from a process. It does this by creating an isolation boundary around a process which prevents the process from affecting or accessing anything outside of its boundary.



In the web context, we sandbox the rendering engine, which runs JavaScript code, from the higher-privilege browser kernel. The use cases of sandboxing can extend to running any application that may be buggy or malicious, to prevent irreparable damage from being done to the system.

3 Different Types of Sandboxes

There exists a number of different ways to isolate systems and untrusted execution. What follows is a brief analysis of some common methods with advantages and disadvantages denoted with a '+' and '−' respectively.

- **Virtual Machine (VM)**: virtualize the hypervisor to configure network and file access for anything that will run in the virtual machine [1]
  - \+ Simple and easy to use
  - \+ Widely applicable to most scenarios
  - \+ Can be used without changing the application
  - − Heavyweight solution (large memory footprint, costs extra CPU cycles, etc)

- **User Level Sandboxes**
  - ∗ **Software Fault Isolation (SFI)**: check that the application is safe to run and isolate possible sources of buggy/malicious behavior (e.g. Native Client [3])
  - ∗ **Safe Languages (e.g. Java, Rust, etc; JS in browser)**: the language handles certain things safely (i.e. memory safety) or has the ability to specify a policy for what it can do
  - \+ Less overhead than a VM
  - \+ Easily extendable under different threat models
  - – Harder to get right (vs VM); SFI could be considered a blacklist approach
  - – Domain specific - an application is only safe if written in a safe language, or has been already been vetted by SFI
- **Physical Isolation**: physical separation of the hardware (i.e. two separate computers with an air gap and no connections between them)
  - \+ Guaranteed to isolate untrusted applications from a trusted system
  - – Not very practical in most use cases
- **Hybrid System Call Interposition**: OS interposes on system calls made by a sandboxed application and checks whether a system call is allowed to be made given the context and arguments
  - \+ Mitigates untrusted application's ability to affect system
  - \+ Easily extendable monitor
  - – Implementation has subtle requirements to be safe

4 Hybrid System Call Interposition Types

There are two main strategies for hybrid system call interposition which are both similar. To highlight their differences and the implications of these differences, a breakdown of the two strategies follows.

4.1 Filtering

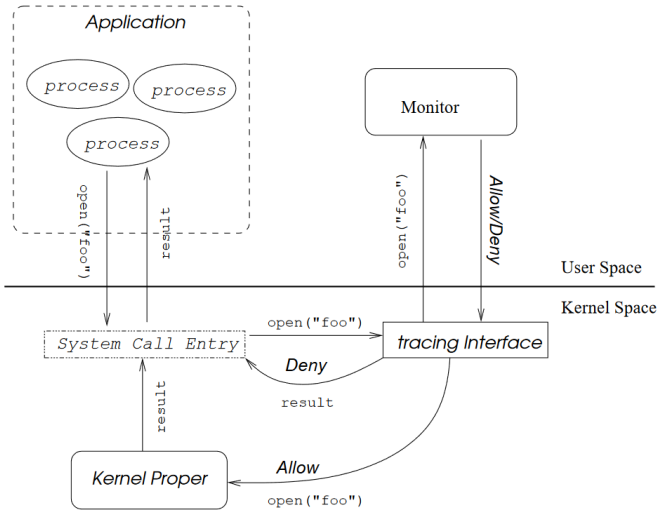


Figure 1: Filtering approach, figure from [2]

The steps are as follows:

1. The sandboxed application makes a syscall (e.g. read/open/write)
2. The OS (tracing interface) intercepts the call and puts the sandboxed application to sleep
3. The monitor checks if the syscall is allowed
    - If yes, then the OS wakes up the sandboxes application which then makes the syscall
    - If no, then the OS returns an immediate error code to the sandboxed application

The trouble with this scheme is that there is a race condition. There is a Time Of Check To Time Of Use (TOCTTOU) vulnerability where the syscall is allowed, but the arguments supplied to the monitor change between the time that they were checked and when they are subsequently used in the syscall. An example of this might be if a user tried to open a symbolic link, which the monitor resolves as a benign file access, but a separate process changes the symbolic link to now reference `/etc/password` before the monitor returns control to us.

4.2 Delegating Architecture (Ostia)

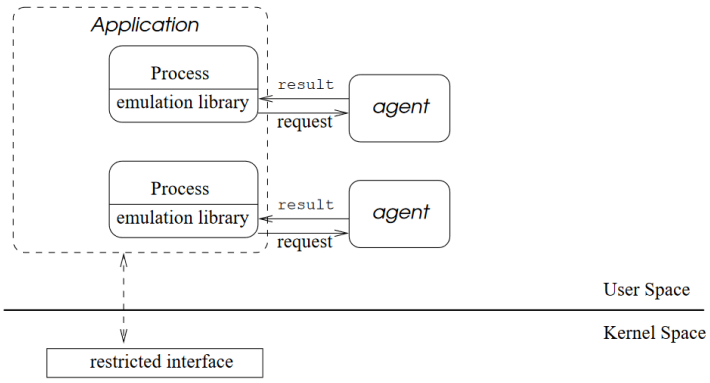


Figure 2: Delegating architecture, figure from [2]

The differences in delegating architecture:

- Virtualize the whole syscall/request as a remote procedure call (RPC)
- User agent executes both the permission check and the syscall for the sandboxed application, then returns the results
    - ∗ This fixes the race condition by copying the arguments from the RPC call and using the same arguments for the permission check as for the syscall itself. This local source of truth prevents any changes to the original arguments after the permission check from affecting the arguments used for the syscall.

This design also derives efficiency gains by reducing the overhead of monitoring all syscalls made by the application, and by not requiring the agent to talk to the system between the permission check and the syscall happening.

5 A Different Example: Native Client (NaCl)

As mentioned previously, Google's Native Client is an example of a user level sandbox, or more specifically software fault isolation; we will dive into how it works to see exactly how it protects users. The motivation for NaCl is that different systems need a centralized method to communicate, but one of the current solutions, browsers and JavaScript, just is not as quick or robust as running native code; namely it can be dramatically slower in many cases. The proposed solution is to run browser applications as native code to speed up execution without compromising the security/isolation that the browser provides for JavaScript or requiring the installation of more dependencies. This then has many possible application domains:

+ Can run existing and legacy applications (no new dependencies)

+ Heavy computation in enterprise applications

+ Multimedia applications

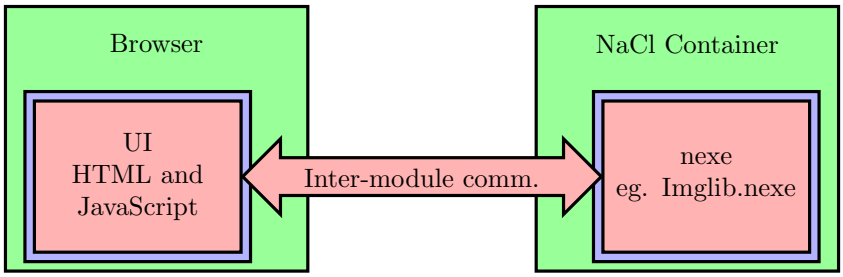
+ Games

+ Any application which requires hardware acceleration

Access to the Native Client gives the ability to take full advantage of the available resources without needing to know the resources a system will have a priori, whereas generic browser applications cannot access a client's full computational potential without the possible security repercussions. The challenge becomes to run applications natively without this compromise in security. Furthering this issue is that for a program to run natively, we need to handle implementation details across many different kinds of systems.

5.1 NaCl - At A Glance

To handle the different system implementations, a Native Client exists for each operating system that we will run these applications on[1]. Developers will then write their code as usual before using the NaCl toolchain to generate an executable (.nexe) which can be interpreted by a local Native Client. The Native Client then acts as a container to isolate the applications running in it, while also providing trusted access to the system's resources.

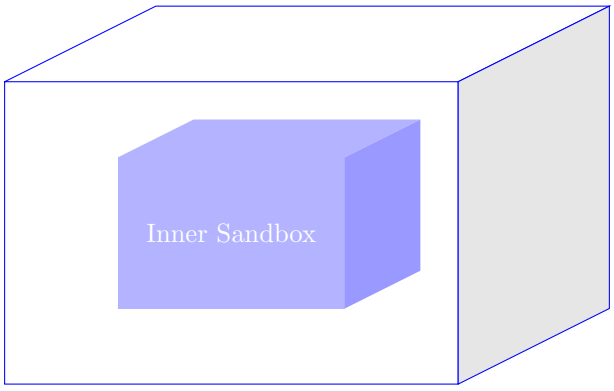


5.2 NaCl - A Closer Look

We don't implicitly trust the nexe binary given to us, but trust NaCl and it's ability to verify that an nexe is safe to run. Part of this trust model relies on the fact that strict constraints are enforced when generating an nexe so that it can be verified quickly before being run in the in the local client; if these constraints aren't observed, then the nexe should be disallowed to run. Google claims that NaCl can handle:

- Untrusted modules from any website
- Memory allocation and additionally spawned threads
- Race condition attacks

These guarantees are met with a nested sandbox:



[1]With the rise of WebAssembly, NaCl has become deprecated on all but ChromeOS

Outer sandbox

- Handles the same threat model as the OS
    - Not strictly necessary considering it is a redundant level of security

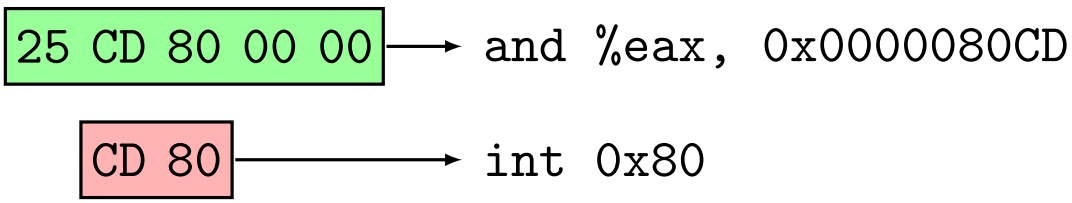
Inner sandbox

- Static analysis to discover security holes in resulting x86
- Disallow self-modifying code and overlapping instructions
- Provide reliable disassembly
    - Necessary for when a local client is verifying new nexe's

5.2.1 Software Fault Isolation

For the static analysis, it is important to identify any potential security threats. Some of these threats are non-negotiable in that access to them means that we cannot guarantee that the application is safe while others depend on exactly how they are handled. Examples of these non-negotiable behaviors are system calls or access to interrupts (no direct access to the OS) and any instructions that can change the x86 state (eg. `lds`, far calls, etc). Other implementation specific sensitive code or protocols are handled by a trusted module, which can be queried through inter-module communication while they are also running in the NaCl container.

Before identifying all cases of the disallowed instructions, NaCl must first identify all the instructions inside of a native code module, but the problem is that because x86 has a variable instruction length, an instruction can be interpreted differently if it is read from the a slight offset:



As you can see, a legal instruction can become one of our disallowed instructions (interrupt) by jumping to it at a different offset, so we have to handle this case. This is where our reliable disassembly comes into play. NaCl imposes alignment and structural rules so that any native code module can be disassembled with all reachable instructions identified. This is done in two steps. First, all `jmp` instructions must jump to a fixed alignment (every byte offset which is a multiple of 32). The new jump semantics are handled by creating a new NaCl psuedoinstruction, `nacljmp`, to replace all jump instructions, so that we apply a bitmask before any indirect `jmp` instruction. Second, every instruction at this alignment must be a valid instruction. With these two constraints, we have handled all possible overlapping instructions and made it possible to reliably disassemble.

5.2.2 Constraints for NaCl Binaries

Once a NaCl binary has been loaded in memory, it is marked as not writable which is enforced by OS-level protection mechanisms during execution. This prevents any possibility of modifying the binary after it has passed the NaCl sanity checks and thus bypassing the security requirements. The binary is statically linked at the start address of zero, with the first byte of text at 64K so that our alignment requirements hold and our memory/code access bounds checking is simplified. All indirect control transfers use a `nacljmp` psuedoinstruction for reasons we outlined above. Similarly for the reasons outlined, the binary contains no instructions or pseudoinstructions overlapping a 32B boundary. All valid instruction addresses are reachable by a fall-through disassembly that starts at the base address so that it can be reliably disassembled and analyzed. All direct control transfers target valid instructions identified during disassembly. Not quite finally, the binary is padded up to the nearest page with at least one `hlt` instruction as a final stopgap to prevent runaway control flow.

References

[1] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, October 2003.

[2] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2003.

[3] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.