CS261 Scribe Notes: Sandboxing

Brian Kim, SID 24332379

1. Sandboxing Background

- Multiple mechanisms for sandboxing, including virtualization, OS-provided sandboxing, syscall filtering, safe languages, binary instrumentation
- Privilege separation vs sandboxing: sandboxing is mechanism for implementing privilege separation
- **Virtualization:** Configure a virtual machine for sandboxing
  - i.e. determine network access and file system privileges
  - easy to use because VMs are easily accessible, so you have instant access to tools for building the sandbox, but there is high memory overhead
  - VMs are also generally heavyweight mechanisms, and also leads to the question of how secure the VM itself is
- **OS-provided sandboxing:** Provide sandboxing features in the OS itself
- **Syscall filtering:** interpose on syscall interface and only allow syscalls that comply with particular policy
- **Safe languages:** languages where the thing you dont want to allow is not even expressible (i.e. no network API)
  - Javascript is an example; the browser is responsible for making sure javascript can not trash your files
- **Binary instrumentation:** take the code of thing you want to sandbox, and modify it, rewrite potentially harmful code so that it becomes helpless

2. Ostia

- Ostia is an example of syscall filtering, which has many pros, including granular control, relatively low overhead, backwards compatability (in contrast to safe languages)
- **Traditional Filtering:** Agent monitors syscalls and blocks them, and all the agent can do is say yes or no for a particular syscall

- Filtering has TOCTOU vulnerabilities where the problem is that the effect of a sys call depends on the state of a system (i.e. the cwd Unix command), such as things like what symlinks resolve to
- Therefore, the following scenario may occur: the agent making the decision looks at the state of the system, decides that the syscall is allowed, but after it decides its allowed, the system state changes (i.e. symbolic link resolution or any mutable state in the system), and the os executes at later time
- The shared global mutable state accessible to multiple entities implies concurrency worries
- **Ostia's proposed filtering fix**: sandboxing app tells the agent what system calls it wishes to allow, but does not have a way to actually do it, so the agent decides if the syscall is allowed, and if allowed, agent does system call
- In particular, the agent can change the state of the system to guarantee atomicity
- Ostia solves concurrency issues: for example, consider an an app that tries to open a relative path (../foo), but if the working directory changes in between, this leads to concurrency issues
- Ostia's fix: if you have a single agent and the agent imposes an ordering and keeps track of cwd, the agent basically can update view of cwd and do other under-the-cover things like convert relative to absolute path
- However, Ostia also has its own share of problems/drawbacks
  - Some syscalls have side effects on processes, so if the agent performs the syscall, side effect happens on the agent and not the actual process
  - The sandboxed app can not bypass the agent and directly invoke the OS, so you need some kernel feature that lets you set a bit on the sandboxed process that doesnt accept system calls that allows you to whitelist a few syscalls
  - **Shadow parsing** is needed; the interface for parsing syscalls is nontrivial; there are two pieces inside the OS, one that decides what to do, and one that actually does performs the syscall
  - The agent needs to emulate same dual feature, so we have to run that code twice, which is dangerous because of two potentially different implementations of the same code (i.e. the agent thinks kernel will do X but kernel will do Y)
  - To solve this problem a general technique called dummification can be leveraged by converting a syscall into multiple steps using a simpler interface, then implement our policy control on that interface, and then figure out what system call to execute
  - In essence, the agent receives a system call, parses it and figures out if it is allowed, and issue a new system call based on its own understanding of what is allowed

  - Another notable example of dummification: parse html into markdown (so there are no html tags), and then check the markdown so that it us syntactically valid and uses only allowed expressions, then finally transform back to safe html

3. SFI Binary Instrumentation

- One problem with syscall filtering is that there is a separate process for sandboxed code, but we may want to allow sandboxed code to run in same process and same address space, like a web browser with plugin code that is resilient so that if the plugin goes haywire the browser is safe
- There are two security features needed so that the plugin code cant overwrite browser code/state, **control-flow** and **memory** safety
- **Control-flow safety** refers to sandboxing the plugin code so that the code can't create a new place in memory and jump to it
- **Memory safety** refers to preventing the plugin code from overwriting browser code or data structures (reading is fine for performance purposes)
- SFI divides the address space into four quadrants: the top (starting from address 0x0) is "unused", the next quadrant contains untrusted sandboxed code, the next qudrant contains sandboxed data, and the final quadrant contains all trusted code/data structures
- The instrumentation comes into play because you can rewrite the sandboxed code by leveraging this memory layout. The following are a few specific examples...
  - `MOV %eax %ebx`

    becomes

    ```
    AND 0x2FFF..F, %ebx
    MOV %eax %ebx
    ```

    This trashes the unused quadrant, which is fine.
  - `JMP 0x12345678 %ebx`

    stays the same.
  - `JMP %eax`

    becomes

    ```
    AND 0x1FF...E0 eax
    JMP eax
    ```

    We can use memory protection so that the first quadrant pages are mapped out correctly.

- Variable-length instructions are annoying, and we can get rid of them by pretending we have a machine where we jump to aligned instructions; thus, jumps have to go to 32-bit aligned addresses
- In the context of SFI, the sandboxed code is pretty useless, so we can use well-designed reference monitors, i.e. the sandboxed code is allowed to call browser code that has a richer API, and we can limit what that API can do
- Dummification can also happen in the context of x86, as we can transform x86 instructions into simpler x86, verified for security, and then transformed back to normal x86