CS 261 - Lecture 8 Notes

Sandboxing

If we have code that we don't trust and may be malicious (i.e. a plugin from a suspicious site), how can we run it safely? We want to run it in a constrained environment, where the untrusted code can run without causing harm to our machine. This is **sandboxing**. There are three main ways to do sandboxing:

- Virtualization - run code in a VM
  - Disadvantages: Very expansive
  - Advantages: Easily generalizable, easy to use
- OS provided sandbox - Mac OS, for example, provides features allowing to run a process with limited permissions
- System call base filtering - software limits what syscalls are allowed (Ostia, Janus)
  - Disadvantage: Suppose we have a browser and we want to install a plugin (i.e. flash), and we are not sure if we trust it. If we want to run it in the same process, Ostia can't help us. It would not be able to tell which process system calls are coming from.
  - Disadvantage: Can end up with fairly complicated rules for filtering
- Safe language - Some languages can't do anything dangerous by design
- Binary instrumentation (SFI)
  - Avoids some of the syscall issues mentioned above

Process based sandboxing

Three challenges:

1. **Shadow state**: in the earliest architecture of such systems, a process would make syscalls, and the sandboxing system (i.e. Janus) would allow or block the syscall. The decision goes to the kernel, which executes the syscall. If the process tries to open a file with a relative path, it depends on the cwd. The kernel maintains the cwd. Say that our sandboxing system (i.e. Janus) keeps track of what cwd as well to be able to process such syscall. But, if it becomes out of sync with the kernel, we have a security issue. What if we have multiple threads? A concurrent thread changes cwd and invokes this security issue. This can allow for example, access to prohibited files. Leads to race conditions, concurrency, and TOCTOU vulnerabilities.
2. **Shadow parse**: syscalls are parsed twice - in the sandbox and kernel. If sandbox parses in a different way than kernel - we have a security vulnerability.
3. **Complex interface**: syscalls interfaces are very complicated.

Ostia solves the shadow state problem.

- Can change arguments to syscalls. It can, for example, canonicalize relative paths.
- **Delegating**: process syscalls are blocked, Ostia gets notified, and Ostia is the one making them, issuing them to the kernel. Ostia can break down a syscall into multiple steps. If there is a complex path, maybe something in the path in a sylink. Ostia

emulates in the user space the file resolution algorithm by breaking it down into components.
    - To combat shadow state - translate syscalls depending on state to stateless.
- Solution for complex interface - dumbification - the syscall API is complex, we will invent a new simple API, translate syscalls to simple interface. We implement the simple interface itself by using underlying OS interface: complex -> simple -> complex

Example: HTML Filter

What if someone sends you JS in HTML email, that will run with all the permissions of gmail (can steal your gmail cookies, etc). Mail providers used to filter HTML emails for markup, and not JS. But today, there are many ways to embed JS in an email. HTML can be parsed to strip JS, but if the parser you are using parses differently than that of your browser, you didn't catch some JS. This is a shadow parsing issue

Google Chrome

Defines three component:
- Untrusted thread: processes can read, write, sigreturn, exit
- Trusted thread: send and receive file descriptors, and allocate memory
- Trusted process (similar to Ostia, delegating piece): if code in the untrusted thread wants a syscall not allowed in the trusted thread, it issues a request to the trusted process, which then provides the untrusted thread what it needs to run. There is also very little reliance on kernel state here.

SFI

With SFI we instrument binary code and make sure it cannot do any harm
- One option - do not allow any syscalls
    - What if I want my code to do syscalls? i.e. video codec - wants to display something on screen. Do Ostia - sandbox code asks browser to do syscall, and browser decides what to do. For example, display code only only on a portion of the screen.
- Second option - we limit the instructions the binary can do - replace illegal instructions. Make sure the code doesn't jump somewhere - code can generate some other code and jump to that region. We need control flow integrity.
- Third option - memory safety - only allow code to interact with specific memory regions. For example,
    - 0x000 -0x0FFFFF: unused
    - 0x100..0 - 0x1FFFF: untrusted code
    - 0x200.000 - 0x2FFFF: data (untrusted data)
    - 0x300 - 0x3FFFFF - trusted code + data + unsandboxed stuff
    - This way the sandboxing checks are a lot easier to make using masking.

- Suppose we see a jump instruction - jump 0x12345678 -> this is fine, you can jump to sandboxed code. No modifications
- What about - jump 0x2345… not allowed
- What about jump %eax - insert a check before to make sure the value is in the allowed range. We can do this efficiently with a mask AND 0x1FFFF0, eax;
- MOV %eax, (%ebx) -> instrument this with a check, we need ebx to be in the range of untrusted data. Use mask AND 0x2FFFFF, %ebx.