Recall

Access control:

- DAC (discretionary access control)
- MAC (mandatory access control)

Today: sandboxing

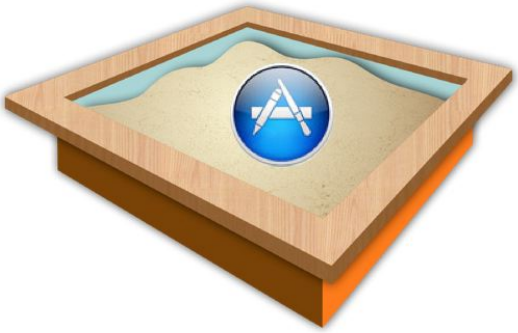
- below the OS: Jails, VMs and containers
- in the OS: capability-based security

Sandboxing

Concept

Reality:

- Processes and virtualization
- DAC and MAC
- Jails, VMs, containers
- Capabilities



Source: Mac Observer

The term *sandboxing* comes from the idea of providing a safe, non-permanent place in which to playfully build and destroy things. Sand castles are fun but impermanent: whatever structure you put into them will be washed away by the waves (another example of entropy at work!).

In computing, a sandbox is a notional container in which software can do whatever it likes, doing some sort of useful work, but ________________ so that any malicious actions will have ________________________________. it's useful to be able to apply such *confinement* or *isolation* to suspect software which may be malicious from inception — this means that ______________________________. However, it's also helpful for software that ______________________________________ – a far more frequent necessity! This does not ______________________, but it can _________________________.

You might well think, "wait a minute, ________________________________"? That's certainly the abstraction we've provided thus far. _____________________ ensures that processes can't directly affect each other. However, ______________________________ break this tidy isolation. Furthermore, they are necessary: what would be the point of ____?

Given that OS abstractions are the weak link here, it's not suprising that people have tried to use OS measures to _______________ programs. We've seen DAC and MAC already, and there are DAC and MAC policies that can provide at least some level of sandboxing. Today we'll also talk about further levels of ____________________ to implement sandboxing, as well as a very different approach that uses a concept called __________________.

DAC-based sandboxing

Unix

- `chroot(2)` and privilege separation*
- Android

Windows (-ish?)

Limitations

Provos, Friedl and Honeyman, "Preventing Privilege Escalation", *Proceedings of the 12th USENIX Security Symposium*, 2003.

One way to prevent applications from accessing OS resources like ________________________ is using Discretionary Access Control.

Since the early 2000s, many Unix systems have used *privilege separation* to limit the damage that can be done by a compromised application that runs with superuser privilege. A program that runs with superuser privilege can use the `chroot(2)` system call to ___________________________ _______________________. Such a program can then ___________________ with `setuid(2)` to a less-privileged user account (historically called `nobody`). In such a state, a program can only access files within its `chroot` area, and it can't do things that only the superuser is allowed to do. This limits the damage that could be done ____ _______________________.

In Android, each application can be assigned a different ________________, allowing that application to prevent other applications from accessing its DAC-controlled resources.

Windows also has a concept of __________________________ that can applied to subjects (processes) and objects (files, etc.) in order to enforce Discretionary Access Control. If you take away a process' SID, it can't access any labeled objects any more!

In all of these cases, however, there are important limitations on ______________________________ _______________. For example, a `chroot`-ed process may not be able to access files outside of its `chroot`, but it can still ____! A Windows process can be prevented from accessing all labeled objects, but when a colleague and I did some Chrome sandboxing work, we found that __________________________ and _______________________

didn't have SIDs! So, an attacker who compromised a "sandboxed" process running as me (e.g., a compartmentalized web browser) would be prevented from accessing my hard drive, but not
______________________________!

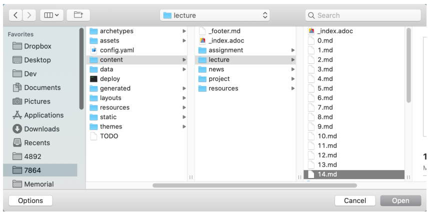
MAC-based sandboxing

Unix

- SELinux, AppArmor
- resource namespaces

iOS / macOS

- MAC framework and "entitlements"
- Also *powerboxes* (not quite MAC)



One can also use MAC to try and sandbox applications. MAC policies can be more expressive than DAC policies, so in theory this can work.

SELinux and, later, AppArmor, brought Mandatory Access Control to Linux. It's possible to write a MAC policy that confines processes quite strictly, e.g., "browser renderer processes can never access the network". MAC is a natural way to express these kinds of ____________________, but ______________________ that refer to *this* or *that* renderer process are much less naturally represented by MAC. SELinux will give one process with the `chrome_sandbox_t` label the same access as another `chrome_sandbox_t`. This is a problem if our goal is to separate my Discord tab, with its complex video handling code, from my Online Banking tab. Also, these policies can be _________________________.

iOS and, later, macOS, brought the FreeBSD MAC Framework into a slightly more popular operating system and used it to provide confinement and isolation of applications. In order to solve the dynamic-access-to-stuff problem, it also brought in the concept of a *powerbox* from a very different access control model: __________________________________.

Problems with DAC and MAC

Dual coding

Privilege

Coherence

DAC and MAC are both very helpful forms of access control within their intended use cases. They are, however, awkward fits for the problem of ______________________ (a more general description of _______________). In addition to the limitations described above, DAC and MAC have more problems in common when applied to ______________________.

Dual coding refers to the problem of ______________________________: first you write code to say ____________________________, then you write a security policy to say _____________________________________. Whenever dual coding occurs, whether it's re-implementations of an algorithm or a code/policy dichotomy, we introduce ___________ __________. In security, mismatches typically cause us to _____________: people jump up and down when you ______________, but when you _____________, the only ones who notice are ________________.

MAC and DAC also require *privilege* to use. Only `root` can `chroot(2)` or `setuid(2)` to limit privilege. Only a system administrator can install a MAC policy. Thus, in order to limit privilege, you have to first ________________! This leads to unhelpful situations in which, e.g., a `setuid-root` binary (a huge security risk) is required in order to implement a security mechanism! ____

Finally, DAC and MAC provide primitives that simply ____________________________ ___________. A poorly-fitting abstraction will be a ___________________, which can cause new problems while solving existing ones. It also spurs _________________ _________: "I have a hammer, now find me something that looks like a nail!"

Jails and VMs

Jail: group of namespace-restricted processes

- BSD *jails*
- Solaris *zones*
- Linux *containers* (though a bit *ad hoc*)

VM: virtual machine

- guest may be *paravirtualized* or host may employ *emulation*

Jails, *zones* or *containers* (e.g., Docker) group a collection of processes together with some common resources. One OS kernal can host many such containers of processes. The processes run as usual, but they are much more limited in terms of what OS objects (files, devices, etc.) they are able to interact with.

Virtual machines take this approach even further, virtualizing not just memory and CPU time (like a process) or an OS kernel (like a jail), but a ____________________. Thus, multiple operating systems can run on top of one CPU. A *paravirtualized* OS kernel is in on the game: it knows that it's running as a regular process within a different OS, and when it wants to do privileged things like tweak virtual memory, it makes a _______________ to the "host" operating system. If your hardware supports virtualization extensions, however, it can run an __________________ _______, trapping on privileged instructions and allowing the host OS to emulate hardware actions like adjusting the guest's virtual memory.

The sandboxing cycle

Source: XKCD 2044

Choose your truth:

- sharing is easy, isolation is hard
- isolation is easy, sharing is hard

Need rigorous isolation... and *controlled* sharing

As in so much else, XKCD spotlights an awkward truth for us...

Capabilities

Means of expressing *principle of least authority*

> *an unforgeable token of authority*

- not a *permission* or *entitlement*
- not a POSIX.1e "capability"
- software-level example: Java references
- system-level example: ???

The *principle of least authority* (sometimes called "principle of least privilege") is the gold standard of confinement, a way of programmatically expressing and enforcing the *ad hoc* "need to know" notion that we explored earlier in the course. Capabilities are a means that *can* be used to express this notion, but you have to get the expression right.

A system-level example of a capability is (*almost!*) is the *file descriptor*. Once you've opened a file, you can interact with ______________________ and perform ______________________ on it. However, the analogy is incomplete, because a normal DAC policy will let you do all kinds of things with a file descriptor. For example, you might want to sandbox a program so that it can only read from a file — OK, you can open it in read-only mode. However, normal Unix DAC will also let you _________________________________ even though you aren't allowed to write to it!

Capability-based systems

Languages: Java to ECMAScript 5

Web: Crypto URLs to WASI

OSes: Capsicum* to CloudABI and Fuschsia

Hardware: GE 645 to CHERI/Morello

* Watson, Anderson, Laurie and Kennaway, "Capsicum: practical capabilities for UNIX", in *Proceedings of the 19th USENIX Security Symposium*, 2010.

Question

Consider a Java object called `HashSet` with methods `add`, `remove` and `contains`. If that object is accessed through a Java interface called `ReadOnlySet`, which only has the `contains` method, will the accessing code be able to modify the set? ______________________________
____

Why capabilities?

Universal mechanism

- vs *ad hoc* mechanisms
- vs privileged mechanisms

Code alignment

... but requires more/better thinking

Rather than one set of mechanisms for separating processes, another set of mechanisms for separating containers and a complete different set of mechanisms for separating virtual machines, capabilities are *meant* to provide a common way of describing objects and things you'd like to do with them. In reality, code-level capabilities, OS-level capabilities and network-level capabilities are implemented differently. However, at least they ____________________ with one another to express _____________________________.

One key element of capability-based access control is the principle that subjects should always be allowed to _______________________________________ without requiring ______________________. That is, if I have read-write access to something, I should always be able to turn that into read-only or write-only access. I shouldn't have to ask permission.

Since capabilities are typically embodied in things like language references, you don't have to separately express __________________________ and ____________________________________.

Sticking to a capability discipline requires thinking carefully about _______________________________________. However, that discipline is the same discipline that you already need to design abstractions with good __________________, good ______________________________ and good _________________.

Summary

Sandboxing

- DAC and MAC
- Jails, VMs and containers
- Capabilities

Beyond sandboxing: compartmentalization