# CSE 127: Introduction to Security

# Memory safety and Isolation

**Nadia Heninger**

UCSD

Winter 2023 Lecture 4

Some slides from Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, David Wagner, Deian Stefan, Dan Boneh, and Zakir Durumeric

# Today

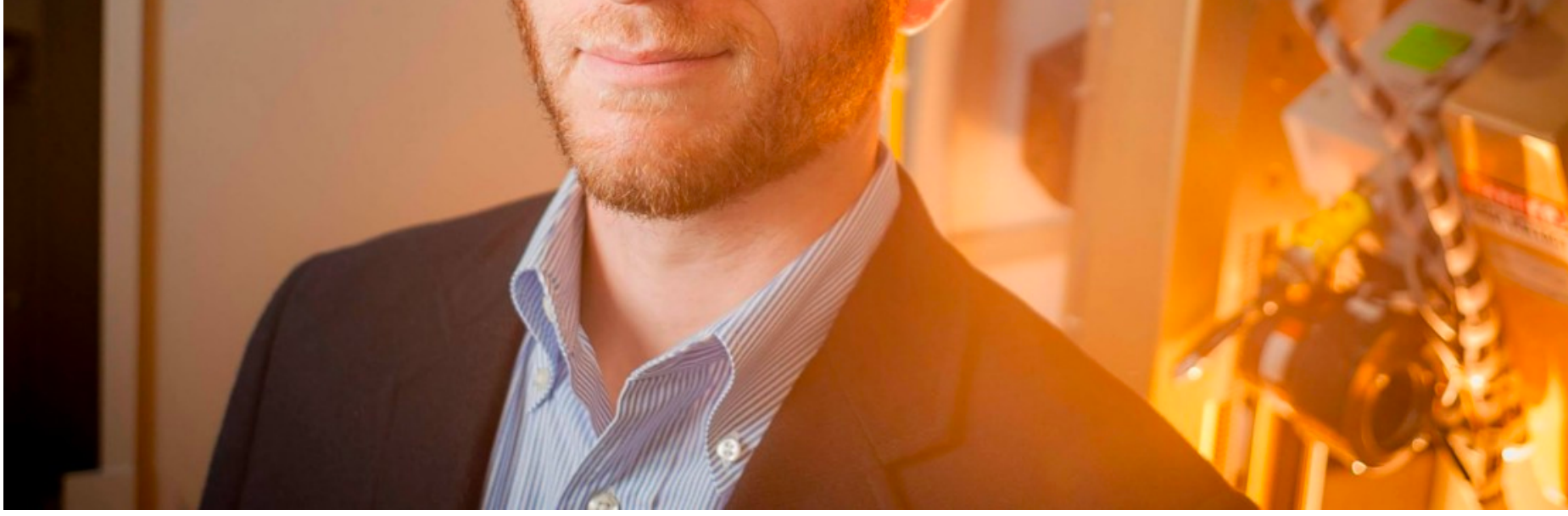
- Return-oriented programming
- Control flow integrity
- Heap corruption
- Isolation

# Last time: return-to-libc

- **Defense:** W^X makes the stack not executable
  - Prevents attacker data from being interpreted as code
- What can we do as the attacker?
  - Reuse existing code (either program or libc)
  - e.g. use `system(“/bin/sh”)`
  - e.g. use `mprotect()` to mark stack executable

Return-to-libc is great, but...

what if there is no function that does what we want?

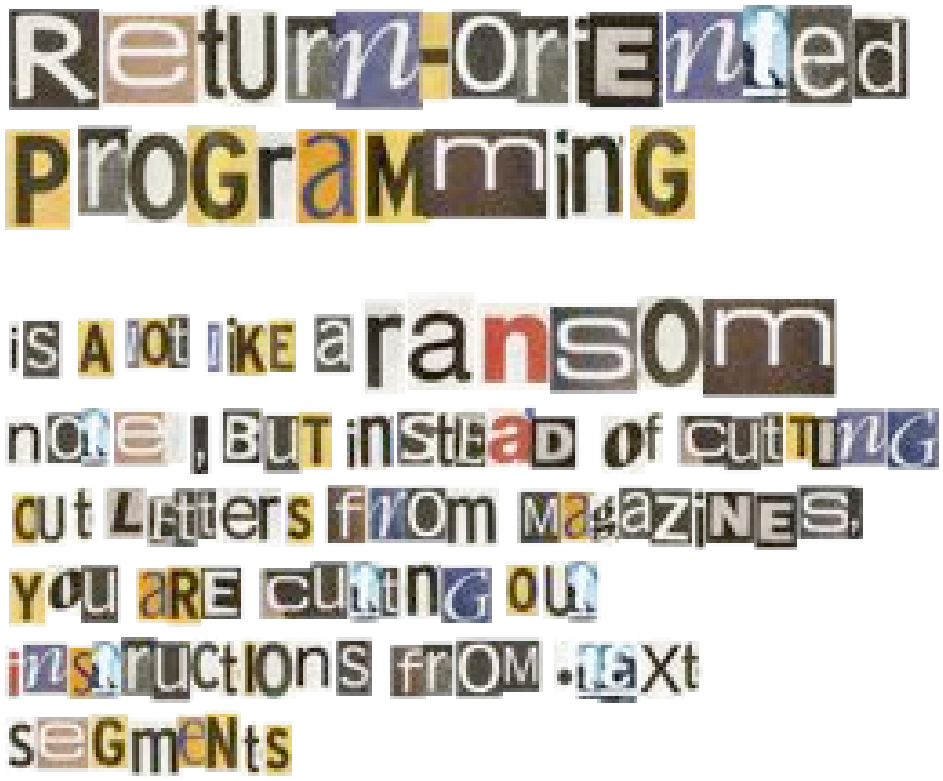


# Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
    - Overwrite saved `%eip` on stack to pointer to first gadget, then second gadget, etc.

# Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
    - Overwrite saved `%eip` on stack to pointer to first gadget, then second gadget, etc.



# Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved `%eip` on stack to pointer to first gadget, then second gadget, etc.
- Where do you often find ret instructions?
  - End of function (inserted by compiler)
  - Any sequence of executable memory ending in 0xc3

```
0000000100002a90        0f b7 7c 24 04 e8 28 0b 00 00 01 c3 89 d8 48 83
0000000100002aa0        c4 08 5b 41 5c 41 5d 41 5e 41 5f 5d c3 55 48 89
0000000100002c30        4f 28 48 8b 46 08 eb 0f 85 c0 45 8b 4f 38 48 8b
0000000100003200        00 48 83 c3 18 48 81 fb a8 01 00 00 75 84 bb 10
0000000100003260        45 89 fd 4c 8d bd b0 f7 ff ff 48 83 c3 18 48 83
00000001000032e0        5b 41 5c 41 5d 41 5e 41 5f 5d c3 48 8d 35 c5 18
0000000100003350        48 83 c4 08 5b 5d c3 48 8d 3d c6 1b 00 00 31 c0
00000001000034a0        c4 70 5b 41 5e 5d c3 e8 a0 0f 00 00 55 48 89 e5
0000000100003550        00 89 d8 48 83 c4 08 5b 5d c3 66 90 7e ff ff ff
00000001000035f0        00 00 00 5d c3 81 c1 00 60 00 00 81 e1 00 f0 00
00000001000036a0        75 06 48 83 c3 10 eb 69 48 8d 7b 68 e8 f7 0e 00
0000000100003740        5e 41 5f 5d c3 55 48 89 e5 41 57 41 56 41 55 41
0000000100003930        5c f0 ff ff 89 c3 48 8d 05 1b 1d 00 00 8b 08 85
0000000100003970        7c 04 85 c9 75 40 41 89 d4 89 c3 48 8d 05 b6 1c
0000000100003990        45 f8 e8 c3 0b 00 00 42 8d 04 2b 23 45 c8 44 89
00000001000039f0        83 c4 38 5b 41 5c 41 5d 41 5e 41 5f 5d c3 31 ff
0000000100003ac0        00 00 48 89 c3 8a 04 1a 88 45 d6 48 83 ca 01 48
0000000100003b00        f8 80 f9 30 75 36 83 c3 d0 41 89 1f 66 bb 01 00
0000000100003b40        9f 80 f9 07 77 08 83 c3 9f 41 89 1f eb 4e 89 c1
0000000100003b50        80 c1 bf 80 f9 07 77 12 83 c3 bf 41 89 1f 48 8b
0000000100003bd0        41 5d 41 5e 41 5f 5d c3 55 48 89 e5 41 56 53 41
0000000100003c30        c6 08 00 00 89 c7 44 89 f6 5b 41 5e 5d e9 e2 08
0000000100003c60        31 c0 48 83 c4 10 5d c3 55 48 89 e5 e8 e9 08 00
0000000100003c70        00 31 c0 5d c3 55 48 89 e5 41 56 53 89 f8 48 8d
0000000100003d10        5e 5d e9 b5 08 00 00 5b 41 5e 5d c3 55 48 89 e5
0000000100003e40        ff ff 4c 89 e6 4c 89 f9 e8 f5 06 00 00 48 89 c3
0000000100003e90        98 00 00 00 5b 41 5c 41 5d 41 5e 41 5f 5d c3 e8
```

# x86 instructions

- Variable length!
- Can begin on any byte boundary!

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =     mov $0x1,%eax
                                  pop %ebx
                                  leave
                                  ret
```

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =     add %al,(%eax)
                                  pop %ebx
                                  leave
                                  ret
```

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =     add %bl,-0x37(%eax)
                                  ret
```

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =     pop %ebx
                                  leave
                                  ret
```

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =          leave
                                       ret
```

# One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3     =          ret
```

# Why `ret`?

- Attacker overflows stack allocated buffer
- What happens when function returns?
  - Restore stack frame
    - `leave = movl %ebp, %esp; pop %ebp`
  - Return
    - `ret = pop %eip`
- If instruction sequence at `%eip` ends in `ret` what do we do?

## relevant code:

```
%eip → 0x08049b62: nop
       0x08049b63: ret
                   ...

       0x08049bbc: pop %e
       0x08049bbd: ret
```

relevant code:

```
          0x08049b62: nop
%eip ⟶    0x08049b63: ret
                ...

          0x08049bbc: pop %e
          0x08049bbd: ret
```

## relevant code:

```
          0x08049b62: nop
          0x08049b63: ret
                      ...

%eip →    0x08049bbc: pop %e
          0x08049bbd: ret
```

relevant code:

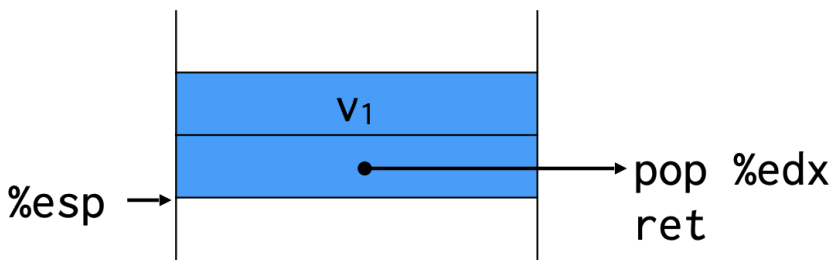
```
          0x08049b62: nop
          0x08049b63: ret
                  ...

          0x08049bbc: pop %e
%eip →    0x08049bbd: ret
```

```
movl v1, %edx
```

# How do you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program
- e.g. if shellcode needs to write a value to `%edx`, use the previous gadget



# Can express arbitrary programs

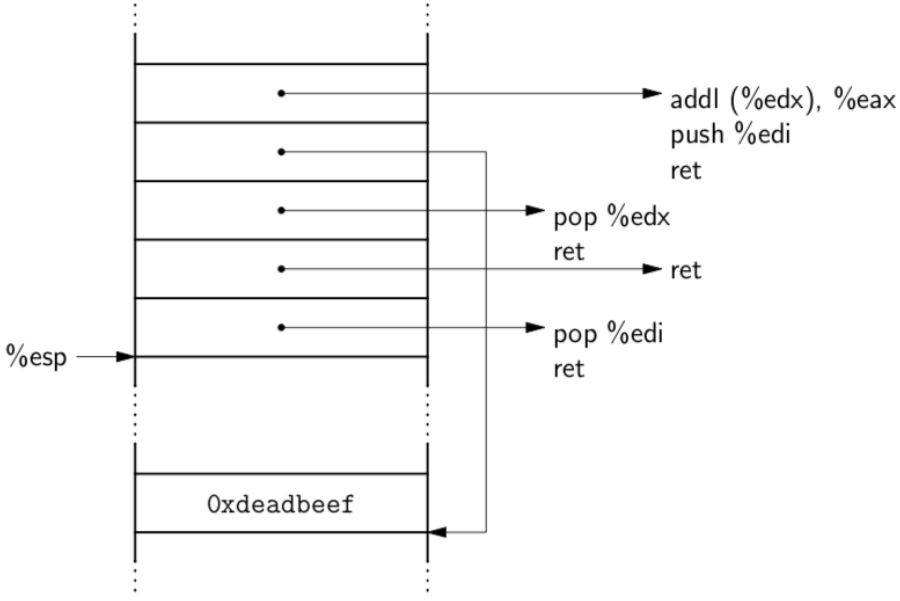



Figure 5: Simple add into %eax.

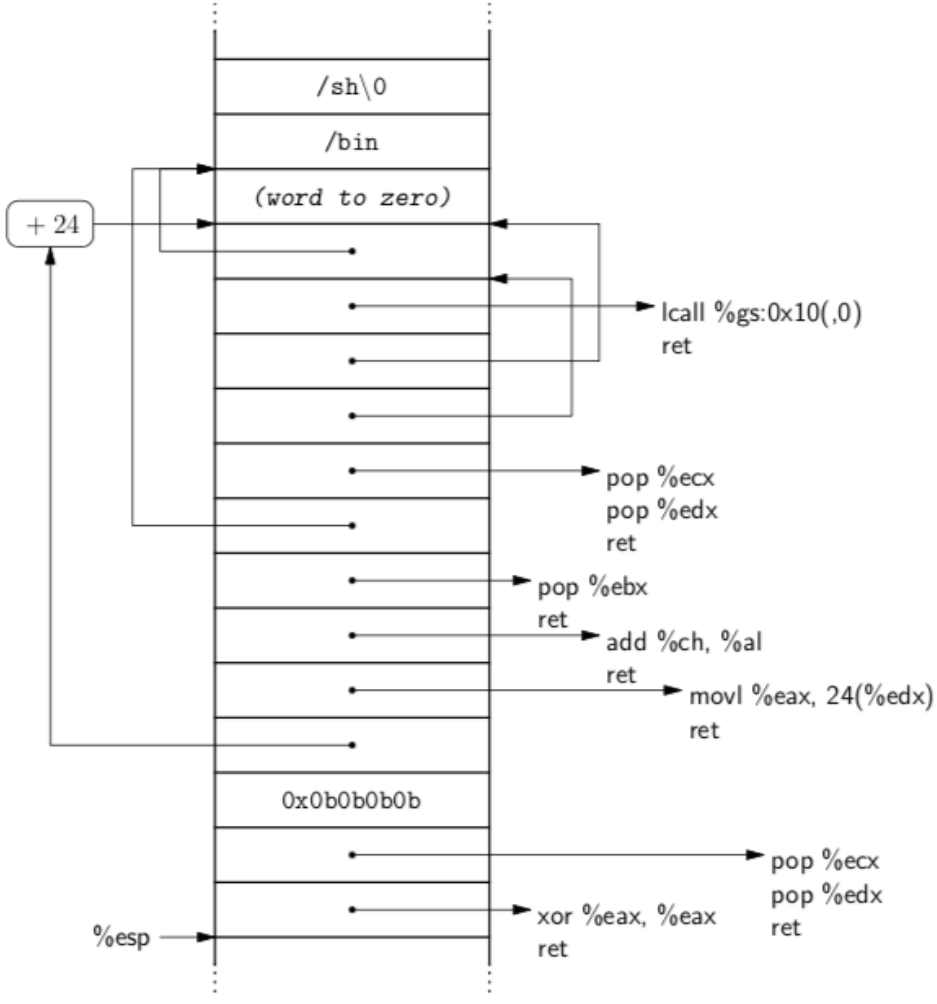



Figure 16: Shellcode.

# Can find gadgets automatically

## Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh

Stanford University

### Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode)/ ARM64
- PowerPC / PowerPC64

# How do you mitigate ROP?

**Observation:** In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses and function pointers)

# Today

- Return-oriented programming

→ Control flow integrity

- Heap corruption
- Isolation

# Control Flow Integrity

- **Idea:** Don’t try to stop the memory writes.
- **Instead:** Restrict control flow to legitimate paths
    - Ensure that jumps, calls, and returns can only go to allowed target destinations

# Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e. direct jumps and calls)?

# Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e. direct jumps and calls)?
    - Address is hard coded in instruction. Not under attacker control.

# Restricting indirect transfers of control

What are the ways to transfer control indirectly?

- **Forward path:** Jumping to or calling a function at an address in register or memory
  - e.g. qsort, interrupt handlers, virtual calls, etc.
- **Reverse path:** Returning from function using address on stack

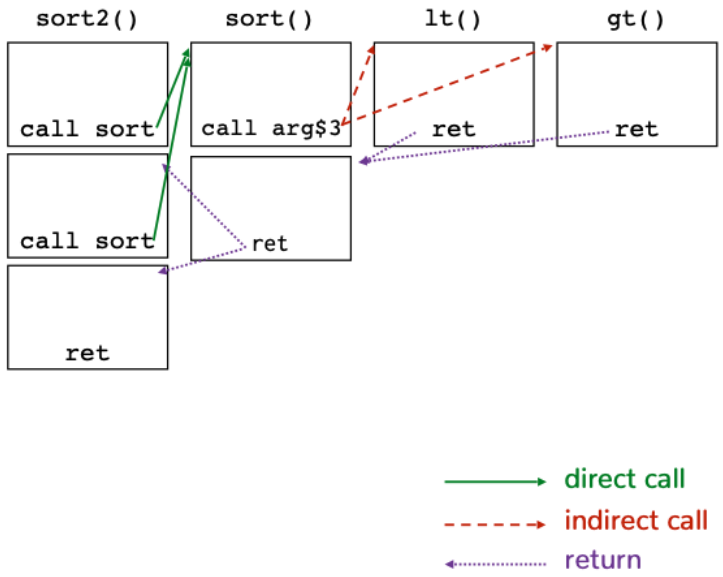
# What's a legitimate target?

Look at the program control-flow graph!

```
void sort2(int a[],int b[], int len {
  sort(a, len, lt);
  sort(b, len, gt);
}

bool lt(int x, int y) {
  return x < y;
}

bool gt(int x, int y) {
  return x > y;
}
```

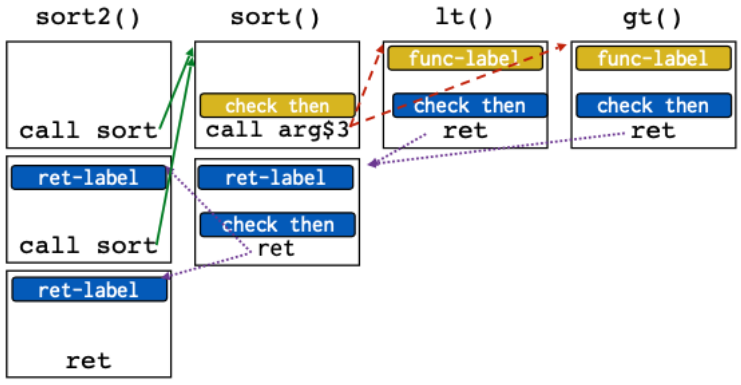


# How do we restrict jumps to control flow graph?

- Assign labels to all indirect jumps and their targets
- Before taking an indirect jump, validate that target label matches jump site
  - Like stack canaries, but for control flow target
- Need hardware support
  - Otherwise trade off precision for performance

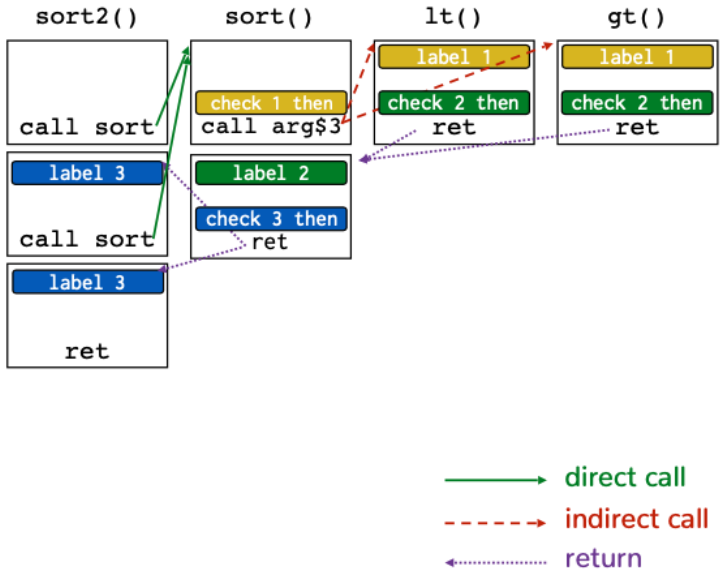
# Coarse-grained CFI (bin-CFI)

- Label for destination of indirect calls
  - Make sure that every indirect call lands on function entry
- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of a basic block



# Fine-grained CFI (Abadi et al.)

- Statically compute CFG
- Dynamically ensure program never deviates
  - Assign label to each target of indirect transfer
  - Instrument indirect transfers to compare label of destination with the expected label to ensure it's valid



# Control Flow Integrity Limitations

- **Overhead**
  - Runtime: every indirect branch instruction
  - Size: code before indirect branch, encode label at destination
- **Scope**
  - CFI does not protect against data-only attacks
  - Needs reliable W^X

# How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
- Coarse-grained CFI can return to many sites
  - Can use a shadow stack to implement fully precise CFI

# Today

- Return-oriented programming
- Control flow integrity

→ Heap corruption

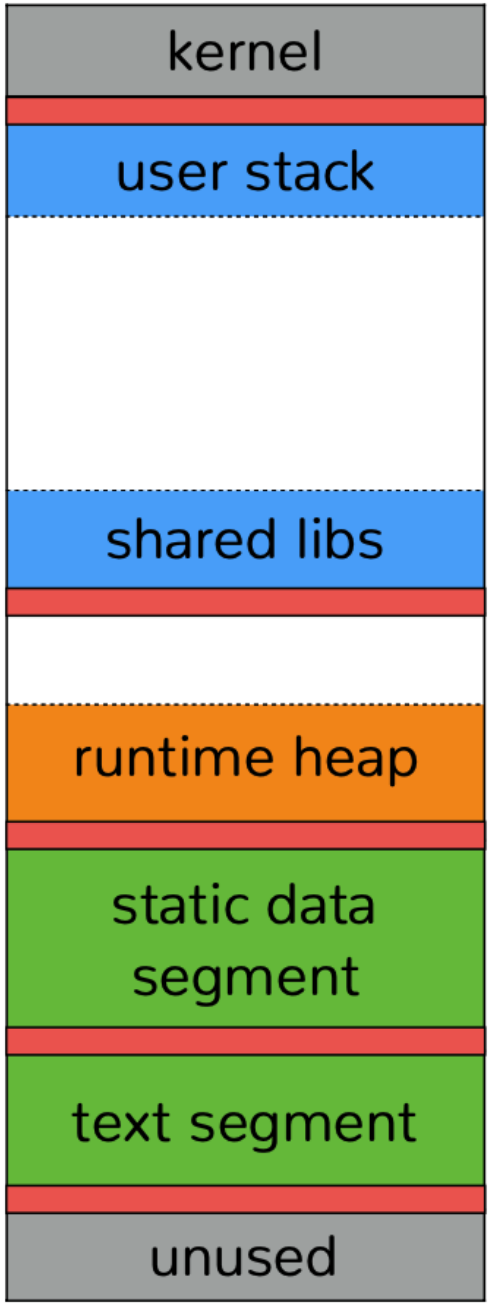
- Isolation

# Memory management in C/C++

- C uses explicit memory management
  - Data is allocated and freed dynamically
  - Dynamic memory is accessed via pointers
- You are on your own
  - System does not track memory liveness
  - System doesn’t ensure that pointers are live or valid
- By default C++ has same issues

# The heap

- Dynamically allocated data stored on the “heap”
- Heap manager exposes API for allocating and deallocating memory
  - `malloc()` and `free()`
  - API invariant: All memory allocated by `malloc()` has to be released by corresponding call to `free()`



# Heap management

- Organized in contiguous chunks of memory
  - Basic unit of memory
  - Can be free or in use
  - Metadata: size + flags
  - Allocated chunk: payload
- Heap layout evolves with `malloc()`s and `free()`s
  - Chunks may get allocated, freed, split, coalesced
- Free chunks are stored in doubly linked lists (bins)
  - Different kinds of bins: fast, unsorted, small, large, …

# How can things go wrong?

- Forget to free memory
- Write/read memory we shouldn’t have access to: Overflow code pointers on the heap
- Use after free: Use pointers that point to freed object
- Double free: Free already freed objects

# Most important: heap corruption

- Can bypass security checks (data-only attacks)
  - e.g. isAuthenticated, buffer_size, isAdmin, etc.
- Can overwrite function pointers
  - Direct transfer of control when function is called
  - C++ virtual tables are especially good targets
- Can overwrite heap management data
  - Corrupt metadata in free chunks
  - Program the heap weird machine

# Use-after-free in C++

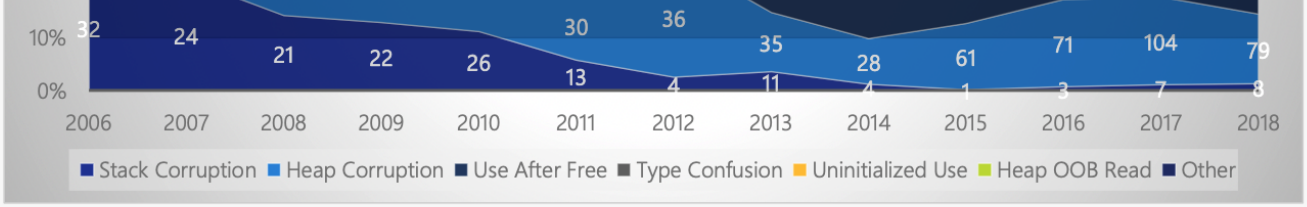
**Victim:** Free object: `free(obj);`

**Attacker:** Overwrite the vtable of the object so entry (`obj->vtable[0]`) points to attacker gadget

**Victim:** Use dangling pointer: `obj->foo()`

# Microsoft Security Response Center (MSRC)

BlueHat IL
February 7th, 2019




Spatial safety remains the mo
common vulnerability catego
(heap out-of-bounds read/wr

Top root causes since 2016:

#1: heap out-of-bounds

#2: use after free

#3: type confusion

#4: uninitialized

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

# Heap exploitation mitigations

- Safe heap implementations
  - Safe unlinking
  - Cookies/canaries on the heap
  - Heap integrity check on malloc and free
- Use Rust or a safe garbage collected language

# What does all this tell us?

If you’re trying to build a secure system,
use a memory and type-safe language.

# Today

- Understand basic principles for building secure systems
- Understand mechanisms used to build secure systems

# Running untrusted code

We often need to run buggy or untrusted code.

# Running untrusted code

We often need to run buggy or untrusted code.

- Desktop applications
- Mobile apps
- Untrusted user code
- Web sites, Javascript, browser extensions
- PDF viewers, email clients
- VMs on cloud computing infrastructure

Systems must be designed to be resilient in the face of vulnerabilities and malicious users.

# Principles of secure system design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Defense in depth
- Keep it simple

# Principle of Least Privilege

- Users should only have access to the data and resources needed to provide authorized tasks

# Principle of Least Privilege

- Users should only have access to the data and resources needed to provide authorized tasks
- Examples:
  - Faculty can only change grades for classes they teach
  - Only employees with background checks have access to classified documents

# Principle of privilege separation

Least privilege requires dividing a system into parts to which we can limit access

- Break system into compartments
- Ensure each compartment is isolated
- Ensure each compartment runs with least privilege
- Treat compartment interface as trust boundary


CLASSIFIED
CLASSIFIED
CONFIDENTIAL
TOP SECRET
TOP SECRET
TOP SECRET


ENTRY RESTRICTED
TO
XCCC AND DXCCC
ON DUTY


EMERGENCY EXIT
Push Until Alarm Sounds
Door Can Be Opened
In 15 Seconds

# Example: Multi-user operating system

In this system:

- Users can execute programs/processes
- Processes can access resources

What's the threat model?

What are the assets?

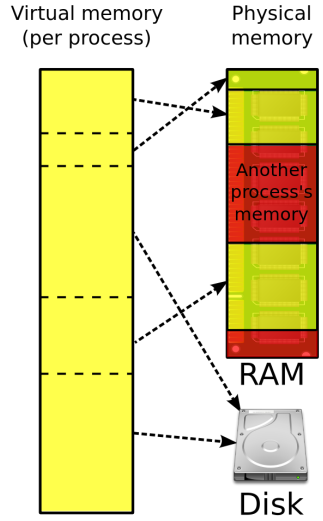
What security properties do we want to preserve?

# Multi-user OS security properties

- **Memory isolation**
  - Process should not be able to access another process's memory
- **Resource isolation**
  - Process should only be able to access certain resources

# Process memory isolation

- How are individual processes memory-isolated from each other?
  - Each process gets its own virtual address space, managed by the operating system
- Memory addresses used by processes are virtual addresses (VAs) not physical addresses (PAs)
  - The CPU memory management unit (MMU) does the translation



# Principle of complete mediation

- Every memory access goes through address translation
  - Load, store, instruction fetch
  - Virtual memory allows address space much larger than physical memory
  - Also means that operating system mediates all process memory accesses and enforces access control policy

# Resource isolation in the Unix security model

In Unix, everything is a file: files, sockets, pipes, hardware devices...

- Permissions to access files are granted based on user IDs
  - Every user has a unique UID
- Access Operations: Read, Write, Execute
- Each file has an access control list (ACL)
  - Grants permissions to users based on UIDs and roles (owner, group, other)
  - root (UID 0) can access everything

# Role-Based Access Control

In a general access control system we can specify permissions in a matrix:

| | hw/ | exams/ | grades/ | lectures/ |
|---|---|---|---|---|
| cse127-instr | r/w | r/w | r/w | r/w |
| cse127-tas | r/w | read | - | r/w |
| cse127-students | read | - | - | read |
| cse-students | - | - | - | read |

# Capabilities vs. ACLs

**ACL:** System checks where subject is on list of users with access to the object.

- Permissions stored by column of access control matrix


**Capabilities:** Subject presents an unforgeable ticket that grants access to an object. System doesn't care who subject is, just that they have access.

- Row of access control matrix

# Unix file permissions are a simplified ACL

```
nadiah@login:/cse/htdocs/classes/wi21/cse127-a$ ls -l
total 32
-rw-rw-r-- 1 nadiah cse127-a-wi 18660 Jan 14 00:34 index.html
drwxrwxr-x 2 nadiah cse127-a-wi  4096 Jan 13 08:42 pa
drwxrwxr-x 2 nadiah cse127-a-wi  4096 Jan 13 19:57 resources
drwxrwsr-x 3 nadiah cse127-a-wi  4096 Jan 14 00:34 slides
```

- Permissions grouped by user owner, group owner, other
- Operations: read, write, execute

# Process UIDs

Process permissions are determined by UID of user who runs it unless changed.

- Real user ID (RUID)
  - Used to determine which user started the process
  - Typically same as the user ID of parent process
- Effective user ID (EUID)
  - Determines the permissions for process
  - Can be different from RUID (e.g. because `setuid` bit on the file being executed)
- Saved user ID (SUID)
  - EUID prior to change

# `setuid`

- A program can have a `setuid` bit set in its permissions
- This impacts fork and exec
  - Typically inherit three IDs of parent
  - If `setuid` bit set: use UID of file owner as EUID

```
-rwsr-xr-x 1 root root 54256 Mar 26  2019 /usr/bin/passwd
```

# `setuid`, `setgid`, and sticky bit

There are three bits:

- `setuid`: set EUID of process to ID of file owner
- `setgid`: set effective group ID of process to GID of file
- sticky bit
  - on: Only file owner, directory owner, and root can rename or remove file in the directory
  - off: If user has write permission on directory, can rename or remove files, even if not owner

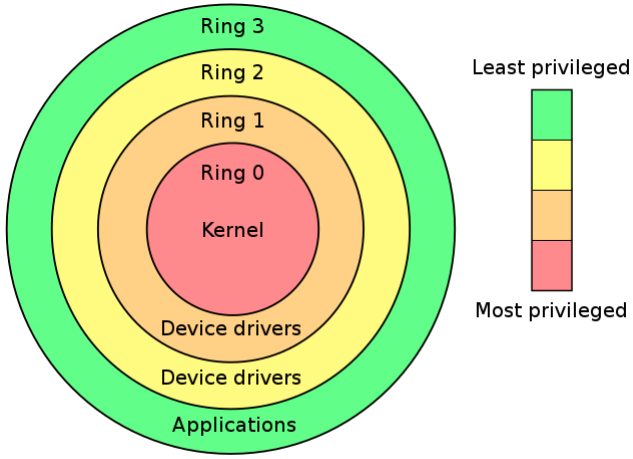
```
drwxrwxrwt  10 root root 12288 Jan 18 20:55 tmp
```

# Overview of Unix file security mechanism

- **Pro**: Simple and flexible
- **Con**:
  - Coarse-grained
  - Nearly all system operations require root access.
  - In practice, common to run many services as root. This violates principle of least privilege and increases attack surface.

# Kernel isolation

- Kernel is isolated from user processes
  - Separate page tables
  - Processor privilege levels ensure userspace code cannot use privileged instructions
- Interface between userspace and kernel: system calls



# Process confinement: system call interposition

**Observation**: To damage a host system (e.g. make permanent changes), an app must make system calls

- To delete or overwrite files: `unlink`, `open`, `write`
- For network attacks: `socket`, `bind`, `connect`, `send`

**Idea**: Monitor app's system calls and block unauthorized calls

# Key component: Reference monitor

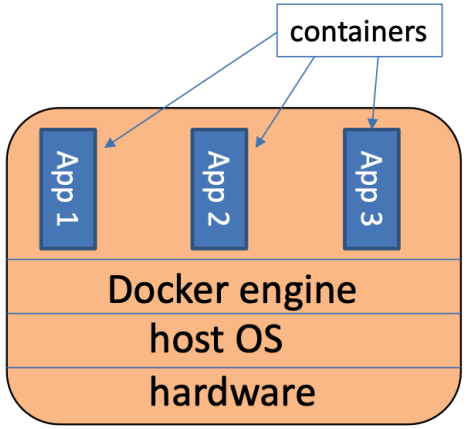
- Mediates requests from applications
  - Enforces confinement
  - Implements a specified protection policy
- Must always be invoked
  - Every application must be mediated
- Tamperproof
  - Reference monitor cannot be killed, or if killed then monitored process is killed too
- Small enough to be analyzed and validated

# System Call Interposition in Linux: `seccomp-bpf`

`seccomp-bpf`: Linux kernel facility used to filter process syscalls

- Syscall filter written in the BPF language
- Used in Chromium, Docker containers…

- Container: process-level isolation
- Container prevented from making syscalls filtered by seccomp-bpf



# Example: Smartphone OS design

Does the threat model for a smartphone differ from a desktop?

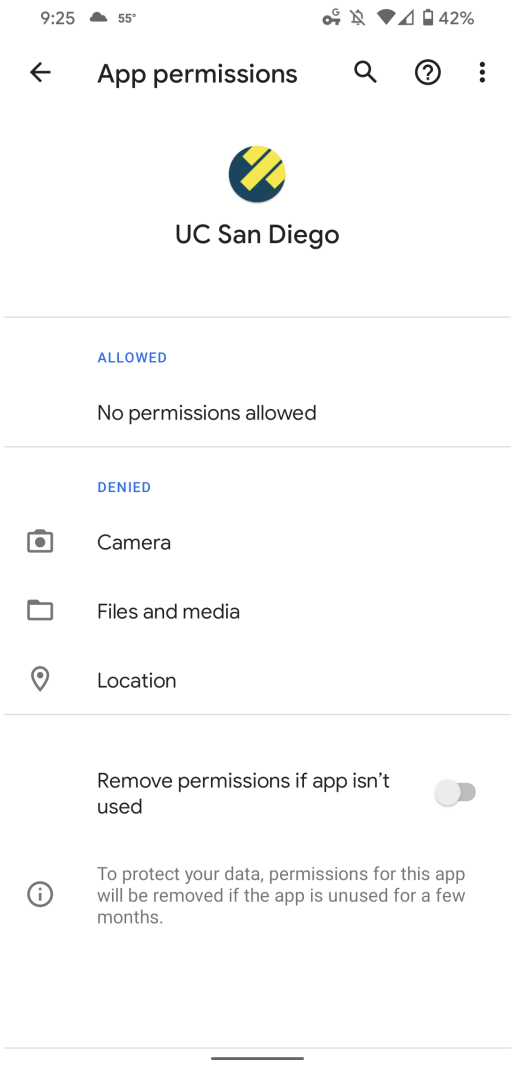
What's the threat model?

What are the assets?

What security properties do we want to preserve?

# Android process isolation

- Android uses Linux and sandboxing for isolation
- Each app runs under its own UID
- Apps can request permissions, which are basically capabilities
- Reference monitor checks permissions on intercomponent communications



# Software fault isolation (SFI)

Placing untrusted components in their own address space provides isolation, but comes with overhead.

Software fault isolation wants to partition apps running in the same address space.

- Kernel modules should not corrupt kernel
- Native libraries should not corrupt JVM

# Software fault isolation (SFI)

Placing untrusted components in their own address space provides isolation, but comes with overhead.

Software fault isolation wants to partition apps running in the same address space.

- Kernel modules should not corrupt kernel
- Native libraries should not corrupt JVM

SFI approach: Partition process memory into segments

- Memory isolation: Instrument all loads and stores
- Control flow integrity: Ensure all control flow is restricted to CFG that instruments loads/stores
- Complete mediation: Disallow privileged instructions
- Syscall-like interface between isolated code

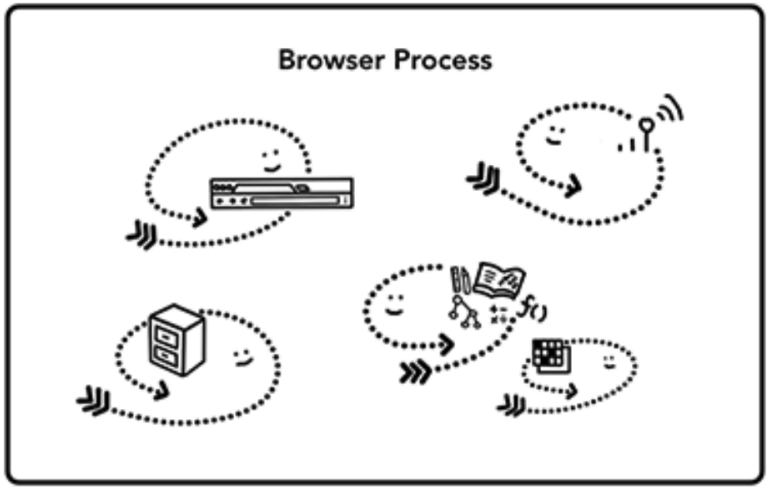
# Example: Browser design

What's the threat model?

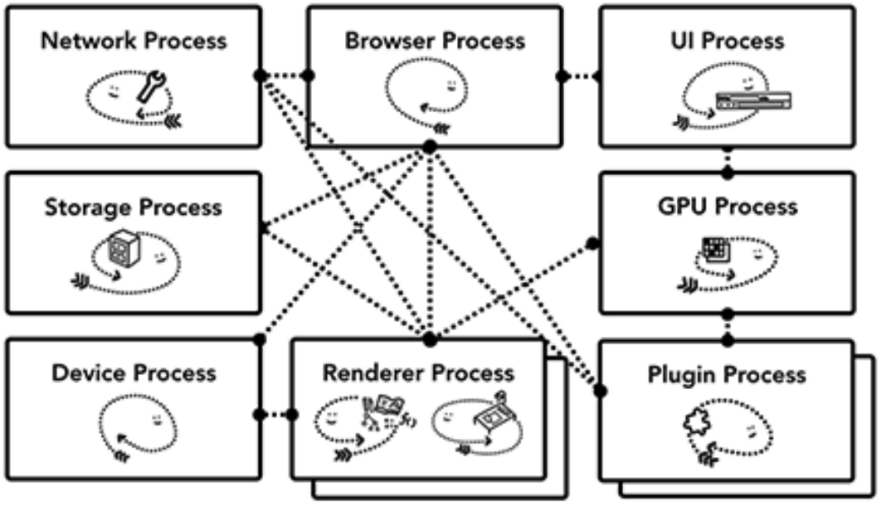
What are the assets?

What security properties do we want to preserve?

# Chrome Security Architecture




Pre-2006



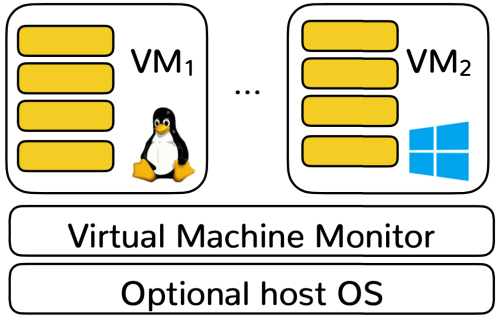

Modern

# Modern Browswer Security Model

- Browser process
  - Handles the privileged parts of browser (network requests, address bar, bookmarks)
- Renderer process
  - Handles untrusted attacker content: JS engine, DOM, etc.
  - Communication restricted to remote procedure calls
- Many other processes (GPU, plugin, etc.)

# Virtual Machines

- Virtual machines allow a single piece of hardware to emulate multiple machines
- Useful for cloud computing and also for isolation
- Intel has hardware support for x86 virtualization: VMM support in hardware so that operating system can be run in ring 0 without requiring VMM intervention for syscalls



# VMs and Isolation

**VM Isolation for the cloud:**

- VMs from different customers may run on the same machine
- Hypervisor tries to isolate VMs to minimize information leaks

**VM Isolation for the end user:**

- Qubes OS: A desktop OS where everything is a VM
- Every window frame UI identifies VM source

# Hardware isolation: Secure enclaves

- Intel Software Guard eXtensons (SGX)
  - Runs trusted code in an *enclave*
  - Enclave memory encrypted and only decrypted in the CPU
  - Can’t be read even by malicious OS
- Why do we want to protect a program against a malicious OS?

# Hardware isolation: Secure enclaves

- Intel Software Guard eXtensons (SGX)
  - Runs trusted code in an *enclave*
  - Enclave memory encrypted and only decrypted in the CPU
  - Can’t be read even by malicious OS

- Why do we want to protect a program against a malicious OS?

  Example applications:
  - DRM (Digital Rights Management)
  - Secure remote computation
  - Protecting crypto keys or sensitive information

# iOS Secure Boot

Apple devices use a secure enclave coprocessor as part of its boot chain.

Hardware-based root of trust: code and code-verifying keys baked into boot ROM (read-only memory).

Each step of the boot process verifies that the bootloader, kernel are signed by Apple.

What are the positives and negatives of this kind of design?

# Physical isolation: Air gap

To ensure that a misbehaving app cannot harm the rest of the system, you could run it on physically isolated system.

What kinds of systems would you do this for?

What are the downsides?

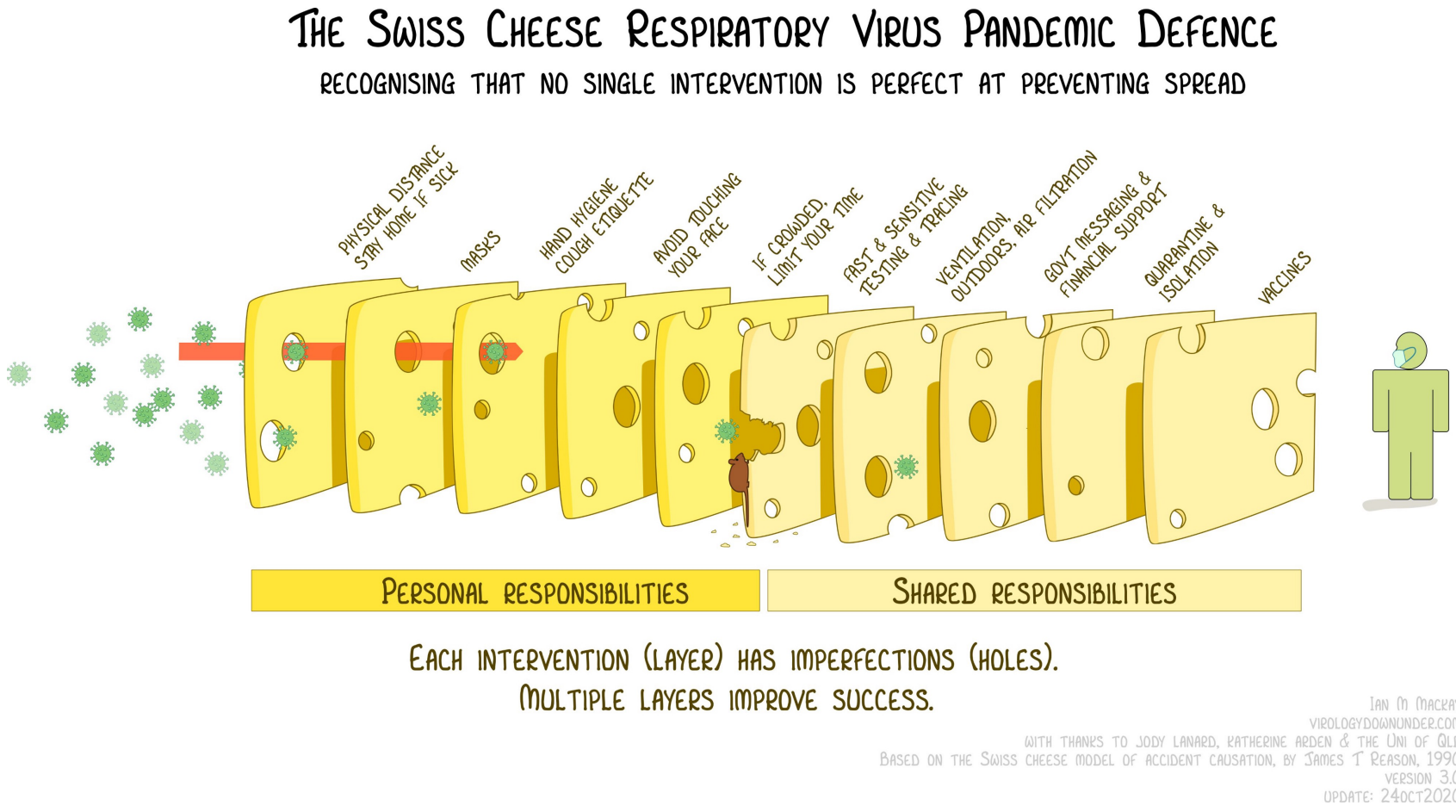
# Principles: Fail closed

What's the problem with failing open?

Why might system designers choose to fail open?

# Principles: Defense in depth

We do not expect any of our defenses to be perfect.



# Principles: Keep it simple

We *have* to trust some components of our system.

In general keeping the Trusted Computing Base small and simple makes it easier to verify.

- In theory a hypervisor can be less complex than a full host operating system.
- A small OS kernel has less attack surface than one with many features.

# Software and hardware isolation techniques

- Memory isolation
- Resource isolation and access control
- System call interposition
- Sandboxing
- Containers
- Virtualization
- Secure enclaves
- Physical air gap

Lesson: Complete isolation is often inappropriate; applications need to communicate through regulated interfaces

# Principles of secure system design

- Least privilege
- Privilege separation
- Complete mediation
- Fail safe/closed
- Defense in depth
- Keep it simple