Lecture 13: Security

CS343 – Operating Systems

Branden Ghena – Fall 2020

Some slides borrowed from:
Tyler Bletsch (NC State), Berkeley CS61C

Northwestern

Today’s Goals

- Introduce OS security considerations.

- Describe memory-based attacks and defenses.

- Explore speculative execution attacks and ramifications.

Outline

- **Design for security**
- Memory attacks and defenses
  - Buffer overflow and No-Execute bit
  - Return-Oriented Programming and Address Space Layout Randomization
- Speculative execution attacks
  - Meltdown
  - Spectre

Why is computer security so important?

- Most public security happens at least in some portion on the honor system
  - Pretty easy to break a window
  - Keyed locks are easy to pick
  - Master keys can be determined and manufactured (Matt Blaze attack)
  - Laws apply after you've done it

Early computers didn't have any security either

- Simple machines for doing computation do not have private files or contention

- Timeslicing machines meant there were multiple users, but all were employees of the same company
  - Permissions needed to be as secure as a file in a locked drawer on a desk

"The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked."
- Ken Thompson, Turing Award Lecture, 1984

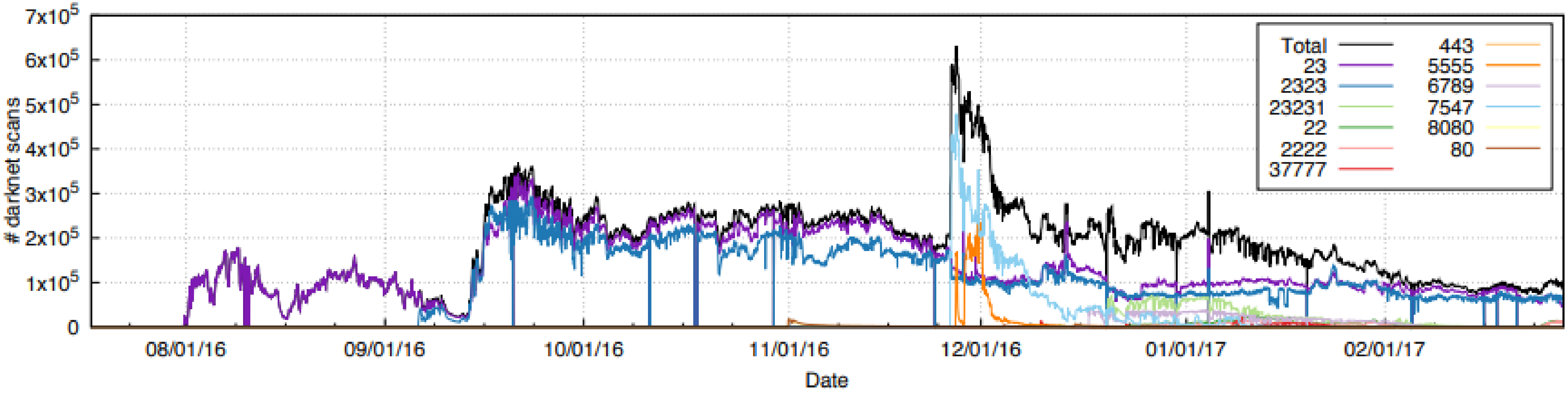
Connectivity of computers makes security a top concern

- Importantly, physical item security is dependent on the fact that one person can only steal one thing at a time
  - And it’s usually obvious when theft occurs

- The internet changed all of this for computers
  - Usually not people breaking into computers manually, one at a time
  - Instead it is computers breaking into computers by means of scripting
  - And you can access a computer from anywhere on Earth

- Breaking into or controlling one car is a crime
  - Controlling 100,000 cars remotely is a problem for the manufacturer

Mirai botnet (2016)

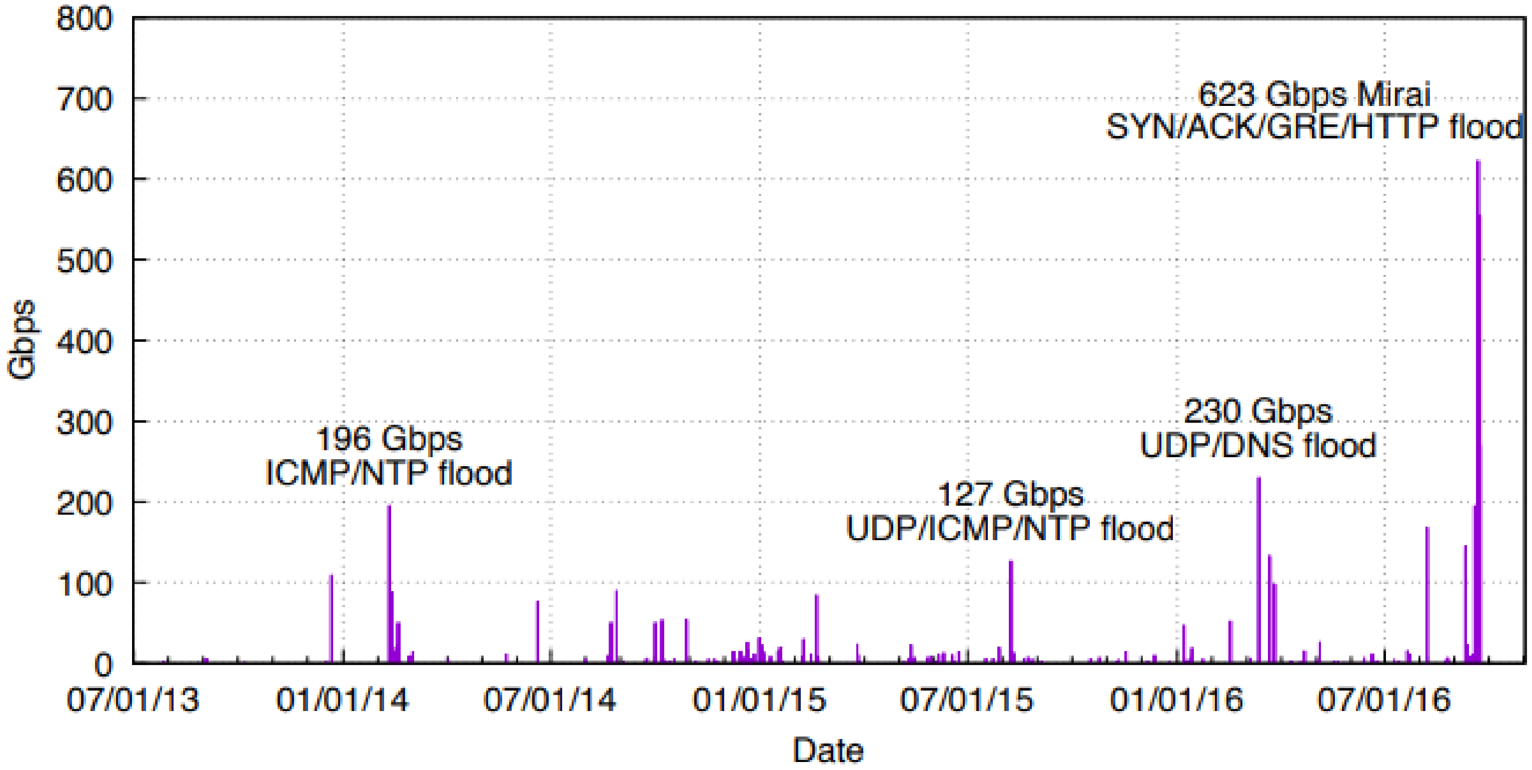


- Takes control of up to 600,000 insecure connected devices
  - IP-attached cameras, DVRs, routers, printers

Botnets can be directed towards denial-of-service attacks

- Mirai is used for DDOS attacks on various websites
  - Krebs on Security blog gets 623 Gbps of traffic during one attack

DDOS attacks targeting Krebs on Security



Trusted Computing Base (TCB)

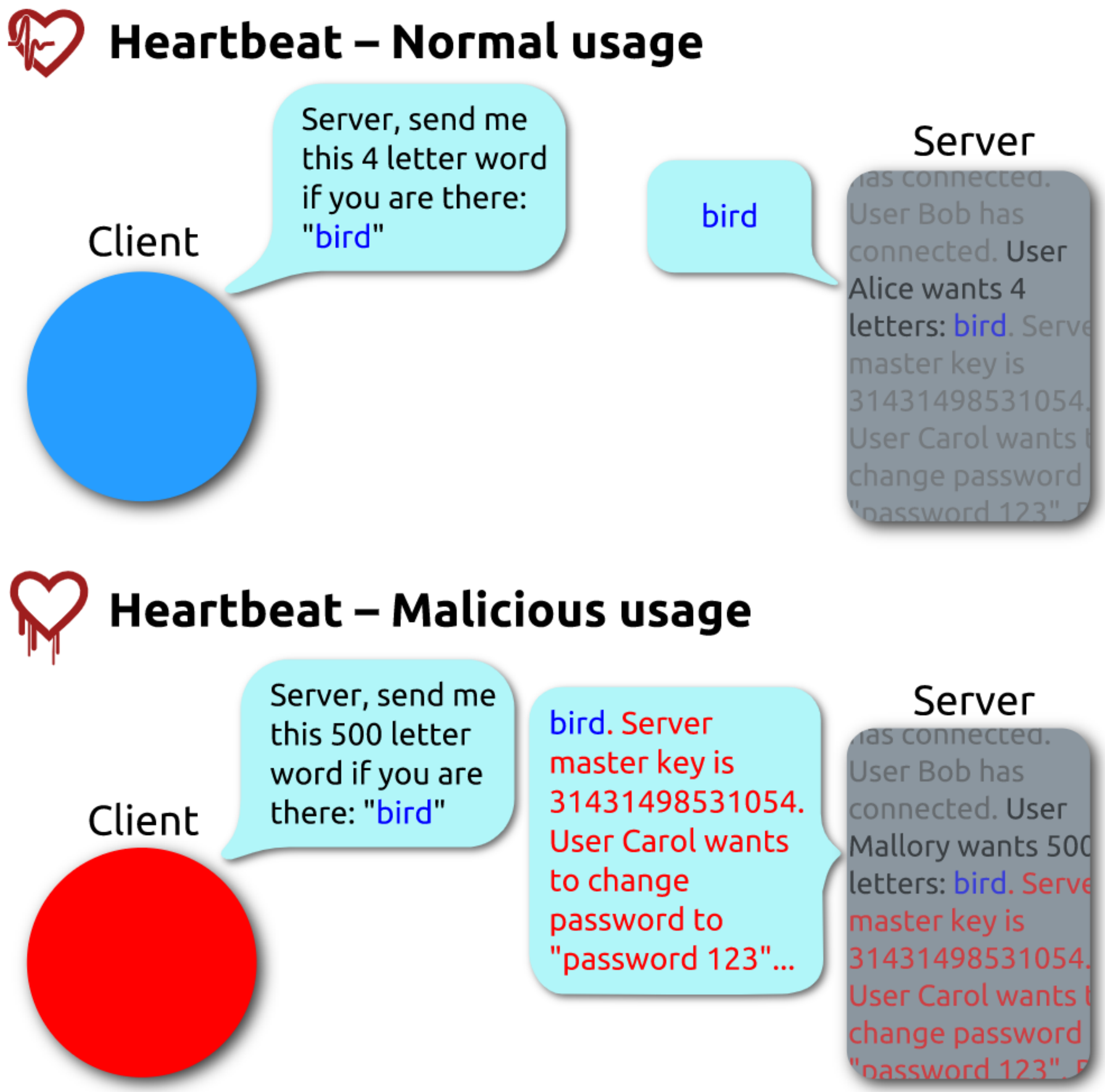
- Trusted Computing Base is everything the OS relies on to enforce security
  - If everything outside of the TCB is “evil”, the TCB can still be trusted
  - Important to be a clear, minimum set of components

- TCB includes
  - Scheduler, Memory Management, Parts of file system, Parts of device drivers

- Anything else must be assumed malicious
  - Processes memory accesses, System call arguments, Received packets

Heartbleed attack

- Vulnerability in OpenSSL
  - 2014
- Started the trend of vulnerabilities with cool names and logos



Modern code bases are enormous

Program/Use Case	Millions of Lines of Code
Unix v1.0	0.01
Average iPhone app	0.04
Space Shuttle	0.4
Windows 3.1	2.5
Mars Curiosity Rover	5
Firefox (2015)	9.7
F-35 Fighter jet	24
Microsoft Office 2001	25
Windows 7	40
Facebook (2015)	62
Debian 5.0 codebase	68

https://www.informationisbeautiful.net/visualizations/million-lines-of-code/

- For many projects, no one person has read and understood all of it
- TCB needs to be agreed upon by everyone working on the project
  - And needs to enforced by everyone in the project

Writing auditable code

- Code style and semantics really do matter!!

- If you want code to be secure, it needs to be read AND understood by many people

- Bad code style/semantics builds up cognitive load of the reader making them less likely to notice when something is wrong
  - 0 versus NULL
  - &buf[0] versus &(buf[0])
  - int x, y, z; versus int x; int y; int z;

Apple “goto fail” SSL bug

Spacing intentional. This code mixes tabs and spaces and has random line breaks.

It is actually decently commented, just not in this particular section.

```
...
 if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
...
```

Apple "goto fail" SSL bug

Spacing intentional. This code mixes tabs and spaces and has random line breaks.

It is actually decently commented, just not in this particular section.

```
...
 if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
...
```

Outside of IF statement!! Always runs.

Security properties OS should enforce

- Confidentiality
  - Private information should remain private
  - Example: processes can’t read memory in another process

- Integrity
  - Mechanisms should not be modified without permission
  - Example: OS data structures can’t be modified by processes

- Availability
  - Resources on the computer should be able to be fairly accessed
  - Example: network access is shared among processes

OS security concerns

- Processor access
  - Integrity: User versus kernel mode
  - Availability: Timeslicing
- Memory access
  - Confidentiality and Integrity: Virtual memory (and permissions)
  - Availability: Swapping
- File access
  - Confidentiality: Permissions (user and group)
  - Integrity: only accessible through system calls

What about devices?

- Device access
  - Confidentiality: User permissions… sort of?
- This gets complicated
  - Should any app I run be able to activate my webcam or microphone?
  - When should Uber be able to access my location?
- Still figuring this one out
  - Smartphones are at the forefront

Security is an arms race

- There is no single fix for system security
  - New attacks are constantly being discovered
  - New solutions are constantly being applied

1. Find a vulnerability and how it can be exploited
2. Fix vulnerability
3. Go back to 1

- But if the OS is designed with security in mind, it’s hopefully harder to find vulnerabilities in the first place

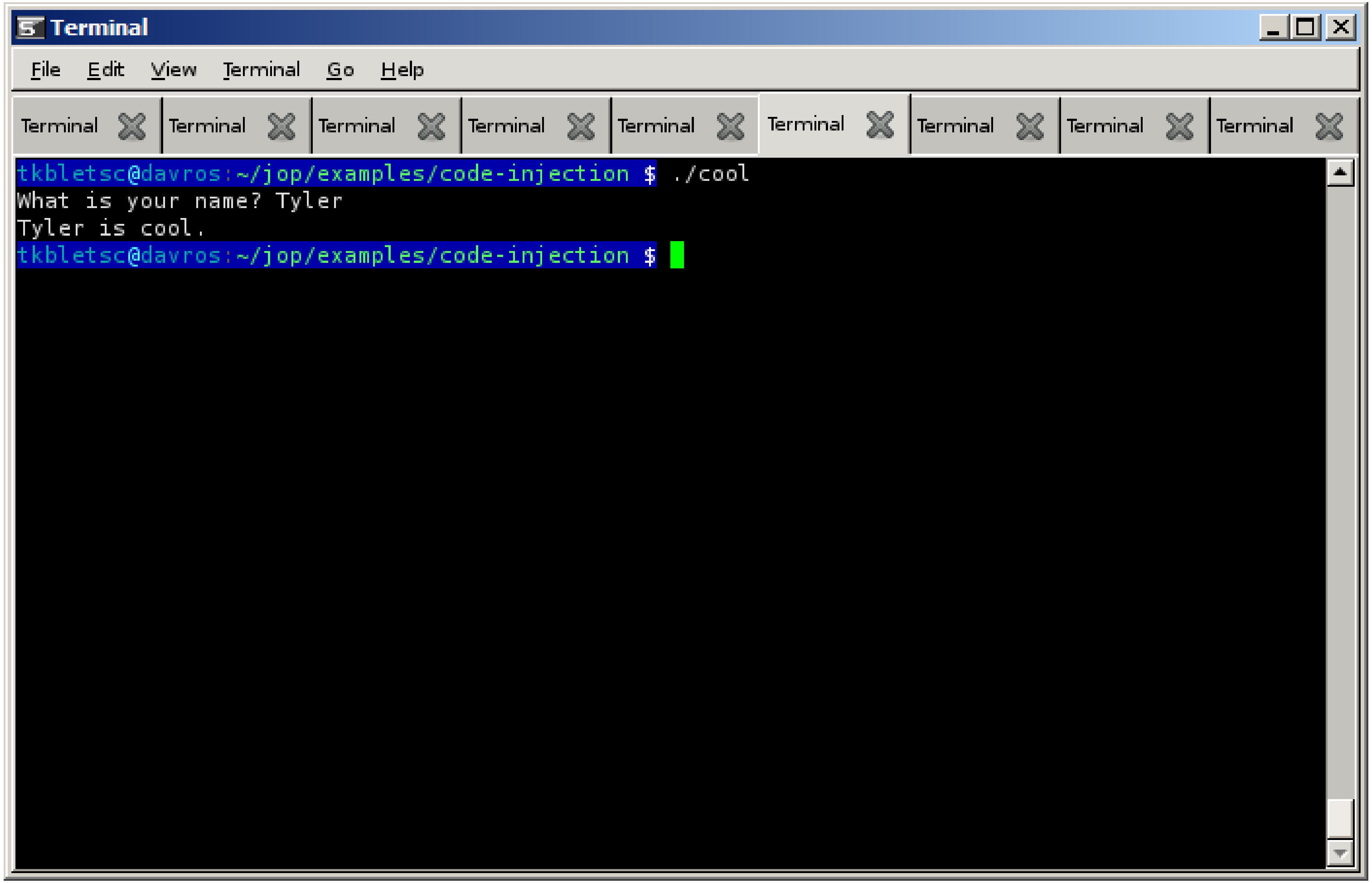
Outline

- Design for security
- **Memory attacks and defenses**
  - **Buffer overflow and No-Execute bit**
  - **Return-Oriented Programming and Address Space Layout Randomization**
- Speculative execution attacks
  - Meltdown
  - Spectre

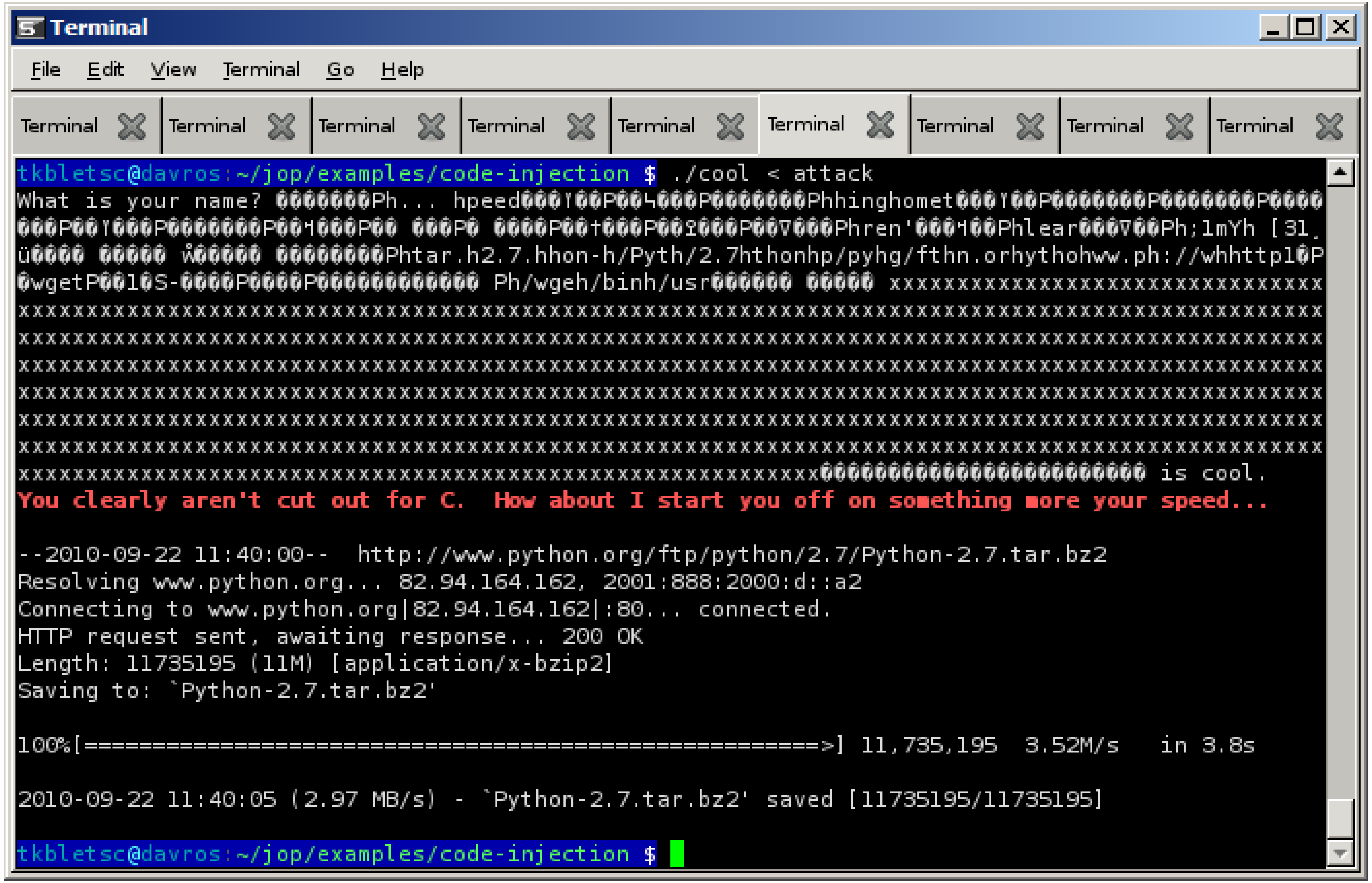
What’s wrong with this code?

```
#include <stdlib.h>
#include <stdio.h>

int main() {
     char name[1024];
     printf("What is your name? ");
     scanf("%s", name);
     printf("%s is cool.\n", name);

     return 0;
}
```

Buffer overflow potential with “nice” input



Buffer overflow potential with “evil” input



Buffer Overflow

- Arrays (buffers) in C are not bounds checked
  - Can keep writing past the end of the array
  - Overwrites either data section or stack section
- Still an incredibly common problem in C
- Key problem
  - Trusting input from an untrustworthy source
  - Users are not part of the trusted computing base
    - Certainly not arbitrary inputs they can make

Unsafe C library functions (and replacements)

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

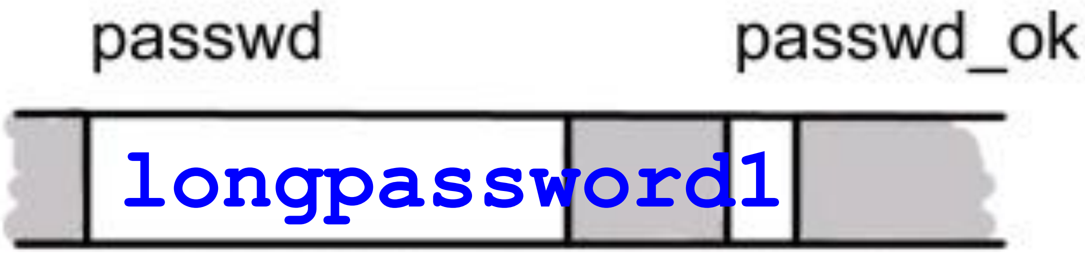
Better choices:

```
char *fgets(char *s, int size, FILE *stream)
snprintf(char *str, size_t size, const char *format, ...);
strncat(char *dest, const char *src, size_t n)
strncpy(char *dest, const char *src, size_t n)
vsnprintf(char *str, size_t size, const char *format, va_list ap)
```

Buffer overflows can overwrite important variables

- Long input string can overwrite variables on the stack
  - Such as the password check

```
int main(int argc, char *argv[]) {
    char passwd_ok = 0;
    char passwd[8];
    strcpy(passwd, argv[1]);
    if (strcmp(passwd, "niklas")==0)
        passwd_ok = 1;
    if (passwd_ok) { ... }
}
```

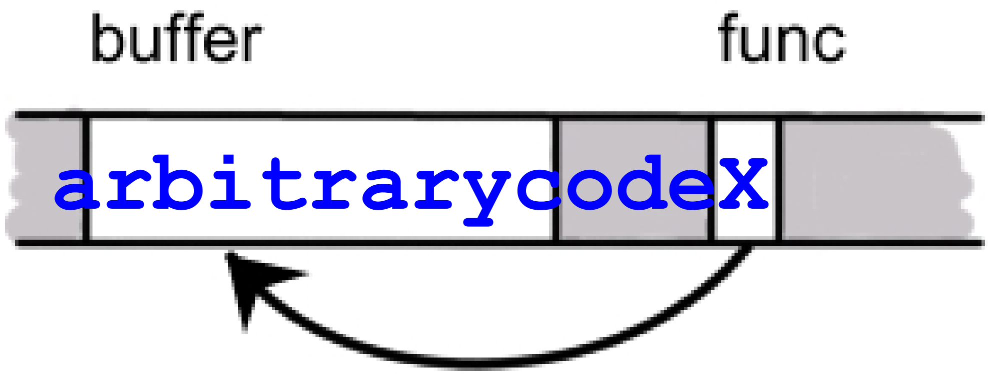


Buffer overflows can overwrite function pointers

- Overwriting a function pointer can allow you to redirect code anywhere

```
char buffer[100];
void (*func)(char*) = thisfunc;
strcpy(buffer, argv[1]);
func(buffer);
```

- First writing machine code in the stack then overwriting function pointer to execute it allows for arbitrary code execution



Return addresses constantly live on the stack

- Recall: When a function is called…
  - parameters are pushed on stack
  - return address pushed on stack
  - called function puts local variables on the stack
- Memory layout

- C's calling convention means arbitrary execution could happen anywhere!

What do you do with arbitrary execution?

- Open a shell that can run anything…

- Top: C code
- Middle: position-independent x86 assembly
- Bottom: machine code hex

```
int main(int argc, char *argv[])
{
   char *sh;
   char *args[2];

   sh = "/bin/sh";
   args[0] = sh;
   args[1] = NULL;
   execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
      nop
      nop                 // end of nop sled
      jmp   find          // jump to end of code
cont: pop   %esi          // pop address of sh off stack into %esi
      xor   %eax,%eax     // zero contents of EAX
      mov   %al,0x7(%esi)   // copy zero byte to end of string sh (%esi)
      lea   (%esi),%ebx     // load address of sh (%esi) into %ebx
      mov   %ebx,0x8(%esi)  // save address of sh in args[0] (%esi+8)
      mov   %eax,0xc(%esi)  // copy zero to args[1] (%esi+c)
      mov   $0xb,%al        // copy execve syscall number (11) to AL
      mov   %esi,%ebx       // copy address of sh (%esi) t0 %ebx
      lea   0x8(%esi),%ecx  // copy address of args (%esi+8) to %ecx
      lea   0xc(%esi),%edx  // copy address of args[1] (%esi+c) to %edx
      int   $0x80           // software interrupt to execute syscall
find: call  cont            // call cont which saves next address on stack
sh:   .string "/bin/sh "    // string constant
args: .long 0               // space used for args array
      .long 0               // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Morris Worm

- November 02, 1988
  - Roughly 88,000 computers on internet at the time
- Worm
  - Invading program that installs itself on additional computers
- Infected several thousand computers, taking down internet for several days

How the worm entered computers

1. Debug vulnerability in *sendmail* - email sending service
    - Connect, enter debug mode, send arbitrary code to execute

2. Buffer overflow in *finger* – lists users on server
    - Send request with more than 512 bytes of arguments
    - Execute /bin/sh

3. Guess passwords
    - Get list of users for the machine worm is already running in
    - Guess username, reverse username, 400 “popular” words, entire dictionary

Effects of Morris Worm

- Morris Worm created too many copies of itself
  - Checked if there was already a worm on the computer before running
  - 1/7 of the executables ran anyways (too high a default)

- Computers ended up with many processes running
  - Check your understanding. How are too many processes harmful?

Effects of Morris Worm

- Morris Worm created too many copies of itself
  - Checked if there was already a worm on the computer before running
  - 1/7 of the executables ran anyways (too high a default)

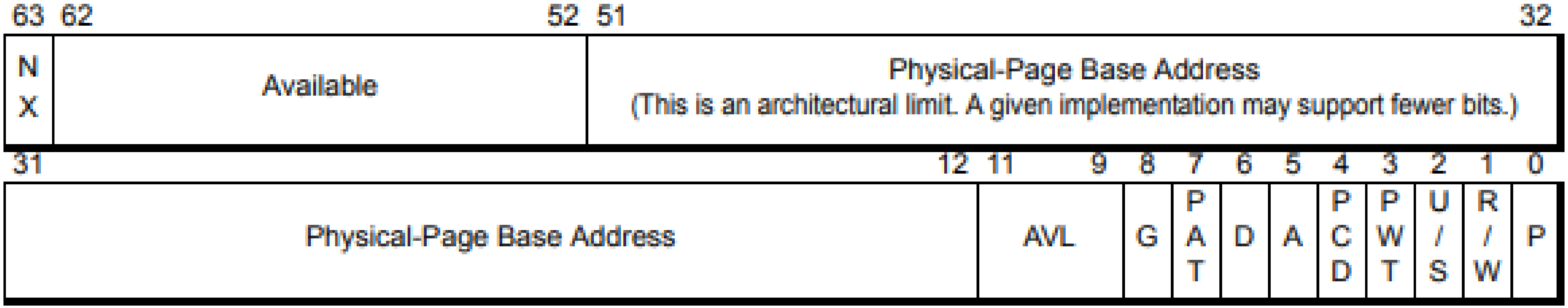
- Computers ended up with many processes running
  - Long response time due to so many processes
  - Thrashing due to too much memory pressure
  - Slowed computers to a halt

- CERT was created to manage software security
- First Computer Fraud and Abuse Act (CFAA)

Disable execution in the stack

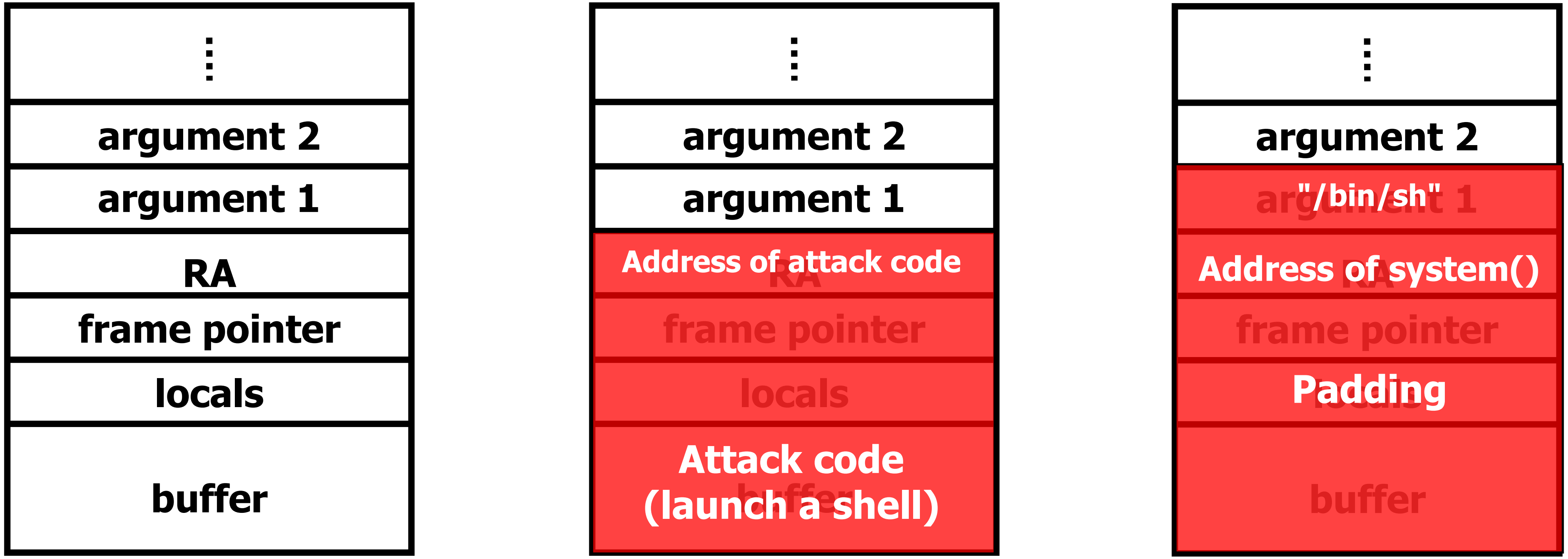
- The OS can allow a region to be written or executed
  - But not both!

- NX bit in x86-64 (no-execute)



Overcoming no-execute

- Do we need malicious code to have malicious behavior? **No**



Default Stack

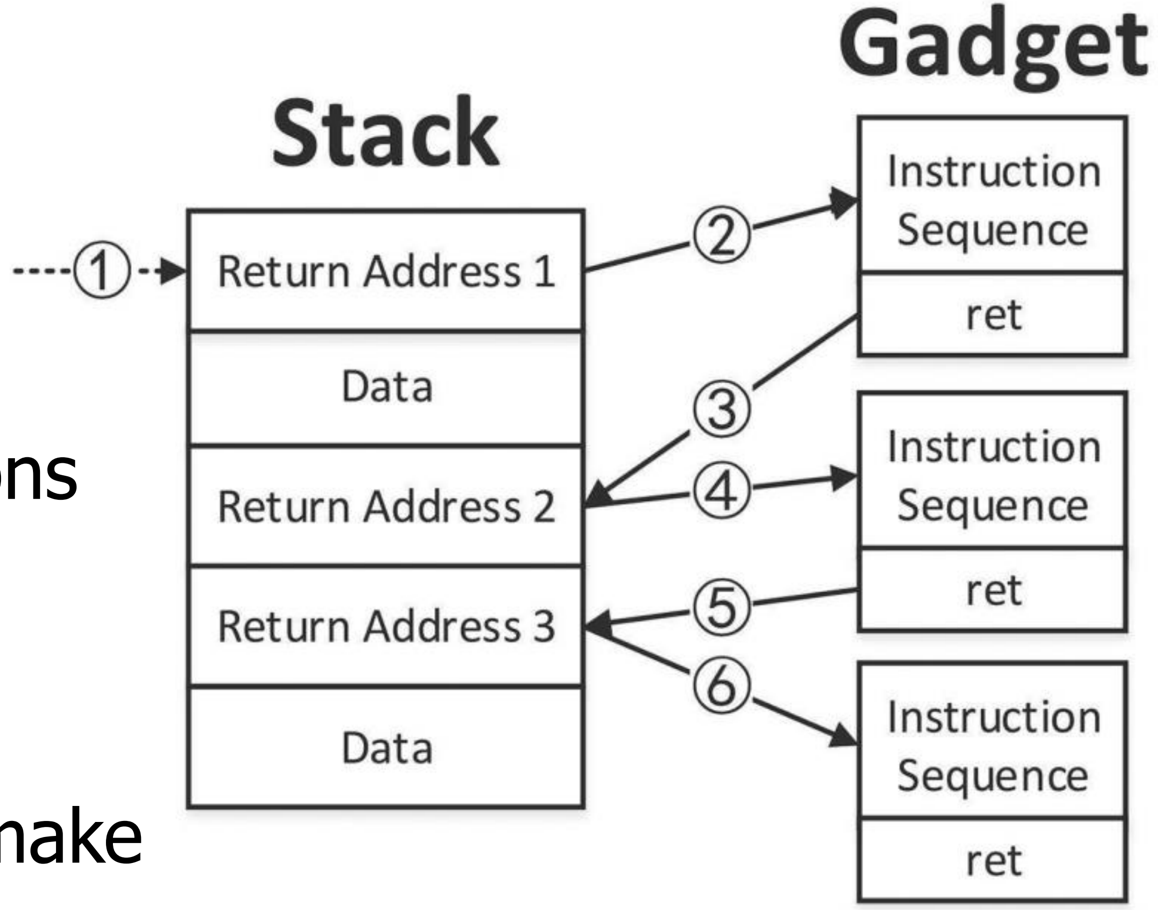
Code injection

Code reuse (!)

"Return-into-libc" attack

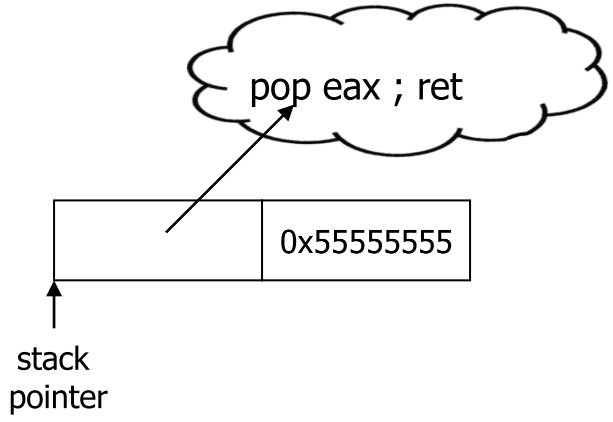
Return-oriented programming

- More general process to enable arbitrary execution without code rewrite
- Look through assembly instructions followed by a return
  - Known as "gadgets"
- Chain these gadget together to make working code
  - By placing addresses on stack

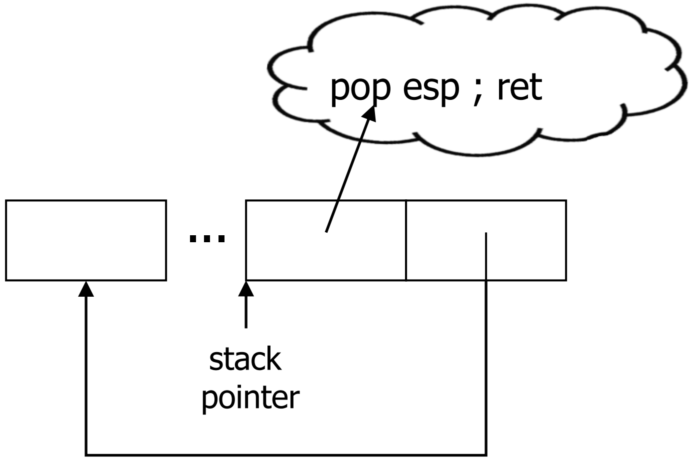


Gadgets can create a Turing-complete programming environment

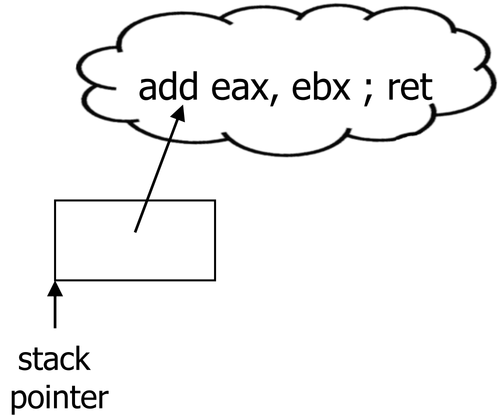
- Loading constants



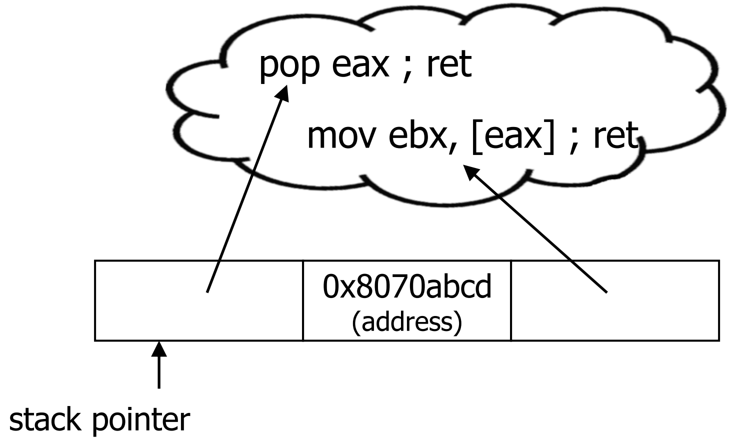
- Control flow



- Arithmetic

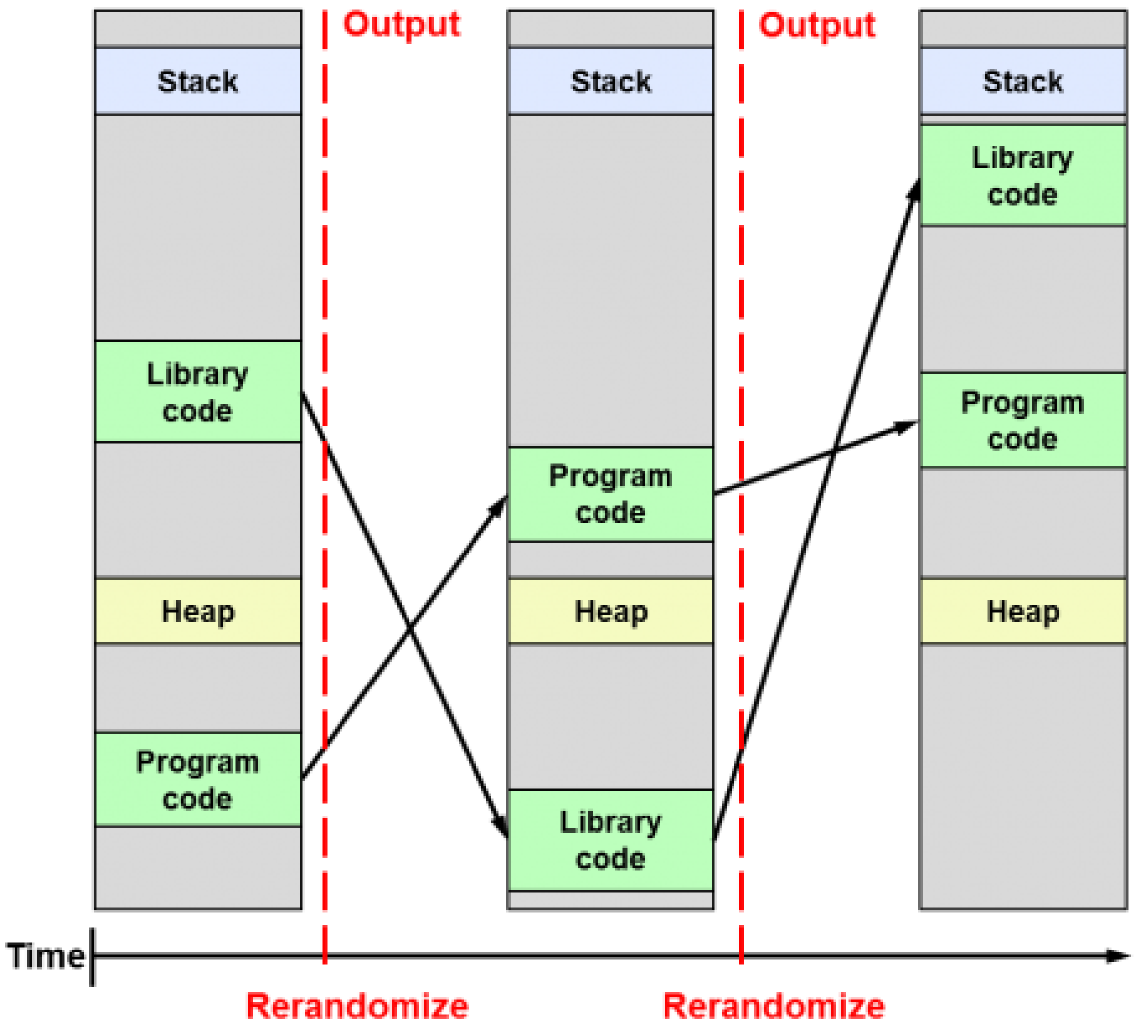


- Memory



Address-space layout randomization (ASLR)

- Randomize memory region locations in virtual memory
  - Already spread throughout physical memory
- Move locations of libraries and code relative to each other
  - Arbitrary address for attacker to send code to gets harder to predict!
- Implemented 2005-2007
  - Linux, MacOS, and Windows
  - 2011 for Android and iOS



Running a process again re-randomizes its layout

Overcoming ASLR

- ASLR is a probabilistic approach, merely increases attacker's expected work
  - Each failed attempt results in crash; at restart, randomization is different

- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - De-randomization attack
    - Just keep trying! (carefully)
    - 32-bit ASLR defeated in 216 seconds
  - Under certain scenarios is less effective
    - Poor source of randomness

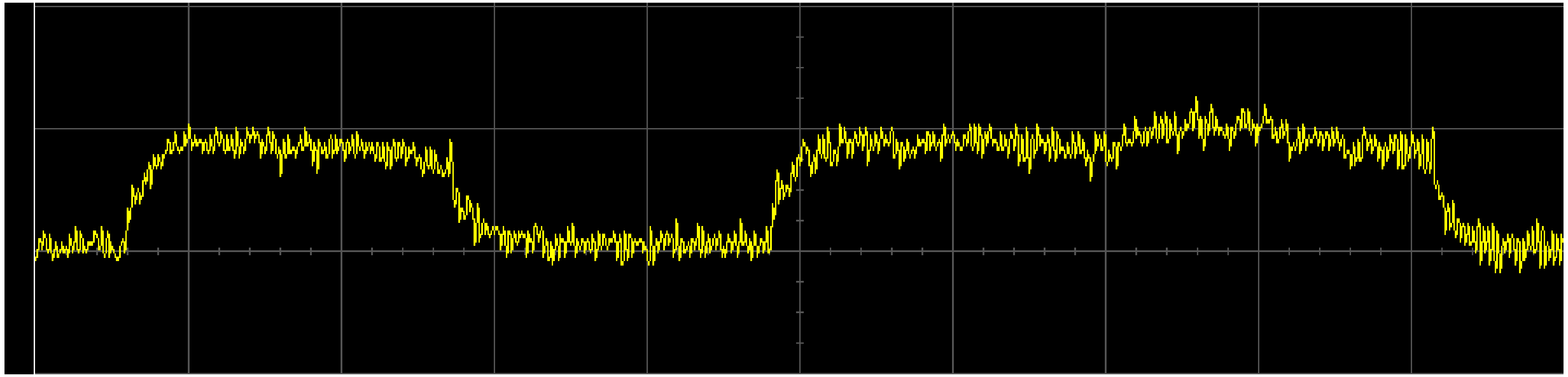
Outline

- Design for security
- Memory attacks and defenses
    - Buffer overflow and No-Execute bit
    - Return-Oriented Programming and Address Space Layout Randomization
- **Speculative execution attacks**
    - **Meltdown**
    - **Spectre**

Side channel attacks

- Important for understanding speculative execution attacks

- Many physical systems have properties that may leak information about internal state
  - Determine RSA key bits based on power use during a decrypt operation
  - Determine length of password by how long it takes to check it



Timing attacks

- Timing attacks can be overcome with constant-time algorithms which always take as long as the worst-case execution time
  - But this means reducing performance

- Caches are essentially one big timing attack
  - Speeds up access to data if it is present in the cache
    - This was the goal!!
  - An attack can know which data was accessed recently

Meltdown

Security vulnerability in all modern processors
that allows arbitrary reads from memory

Disclosed in January 2018 by:

- Jann Horn (Google Project Zero),
- Werner Haas, Thomas Prescher (Cyberus Technology),
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)

Details:

- https://hackernoon.com/a-simplified-explanation-of-the-meltdown-cpu-vulnerability-ad316cd0f0de
- https://meltdownattack.com/meltdown.pdf

Reminder: Speculative Execution

Modern processors have long, complex pipelines

- More than the 5 stages we learned
- Can execute some instructions out-of-order
- Wants to always be doing something

So they are often “speculatively executing” instructions

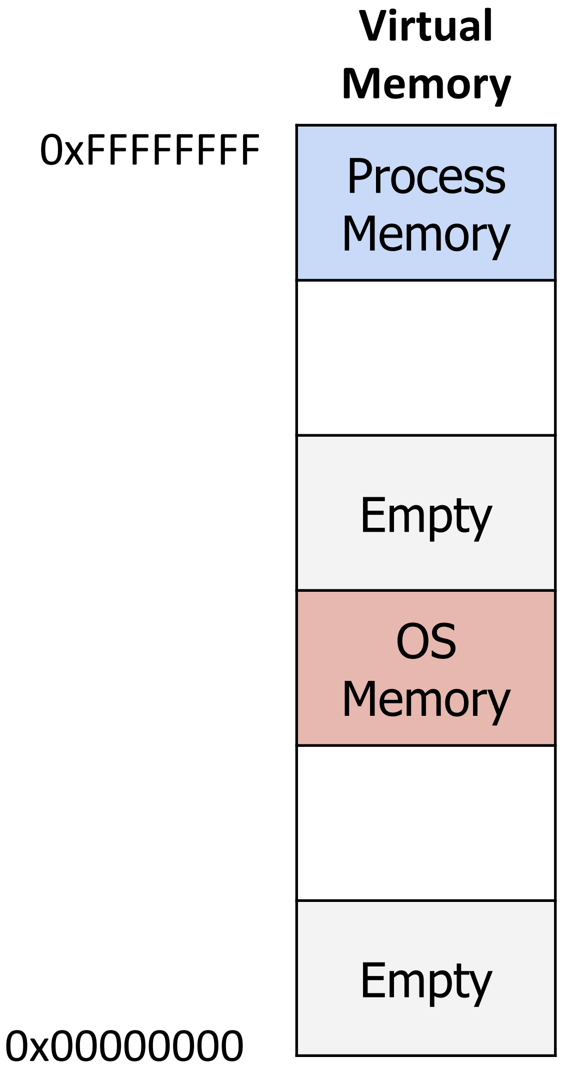
- Perform the operation and throw out the result if we shouldn’t actually do it
- For example, branch prediction

Optimization: Kernel Mapped in Virtual Memory

Page tables map virtual memory to physical memory for a process

But actually, we often leave the OS memory in the page table too…

- Each page is marked as no-read, no-write
- Faster to switch back to the OS
  - No need to TLB flush or page table swap if the OS intends to go right back to process



Step 1: Read from a kernel address

```
mov $KERNEL_ADDRESS_OF_SECRET, %r12
mov (%r12), %eax
```

`%eax` now holds a byte of memory that we shouldn’t able to access

- This will be an invalid page fault!
- Once the instruction actually hits the end of the pipeline...

Step 2: Read based on secret

```
mov $KERNEL_ADDRESS_OF_SECRET, %r12
mov (%r12), %eax
mov MY_ARRAY(%eax), %edx
```

`%edx` is a valid read from our own memory

- This is never going to finish either because the process will have an exception from the prior instruction, but it will start executing...

Step 3: Handle the Exception

```
 mov $KERNEL_ADDRESS_OF_SECRET, %r12
 mov (%r12), %eax
 mov MY_ARRAY(%eax), %edx
```

The processor realizes you tried to read from memory you didn't have access to and generates an exception

- You can catch these and recover
- The invalid instruction and ones after it are rolled back as if they never happened

Everything's still safe right?

The processor never saved any results from the invalid accesses to memory in registers

- So there's no problem, right?

We forgot about the cache

The load affected the cache!!!

```
mov $KERNEL_ADDRESS_OF_SECRET, %r12
mov (%r12), %eax
mov MY_ARRAY(%eax), %edx
```

The value at address `MY_ARRAY+%eax` was saved in our cache

Step 4: Time loads from memory

```
for (int i=0; i<255; i++){
    start_time = time();
    int temp = MY_ARRAY[i*CACHE_BLOCKSIZE];
    stop_time = time();

    if ((stop_time-start_time)==short_time){
        secret = i;
    }
}
```

The cache speeds up the access to the one memory address that was cached due to speculative execution

Step 5: Repeat and Profit

- Now we know the value of a single byte
- But we can repeat this process over and over to read arbitrary memory
  - Read from memory at ~500 kbps

- Incredible part is how relatively simple this attack is
  - Does require systems knowledge of multiple domains
  - Computer architecture, OS, and security

How do we fix this?

1. Stop speculatively executing
   - Already in the hardware
   - Would slow all computers down a lot

2. Stop caching speculative loads
   - Already in the hardware
   - Would slow all computers down a lot

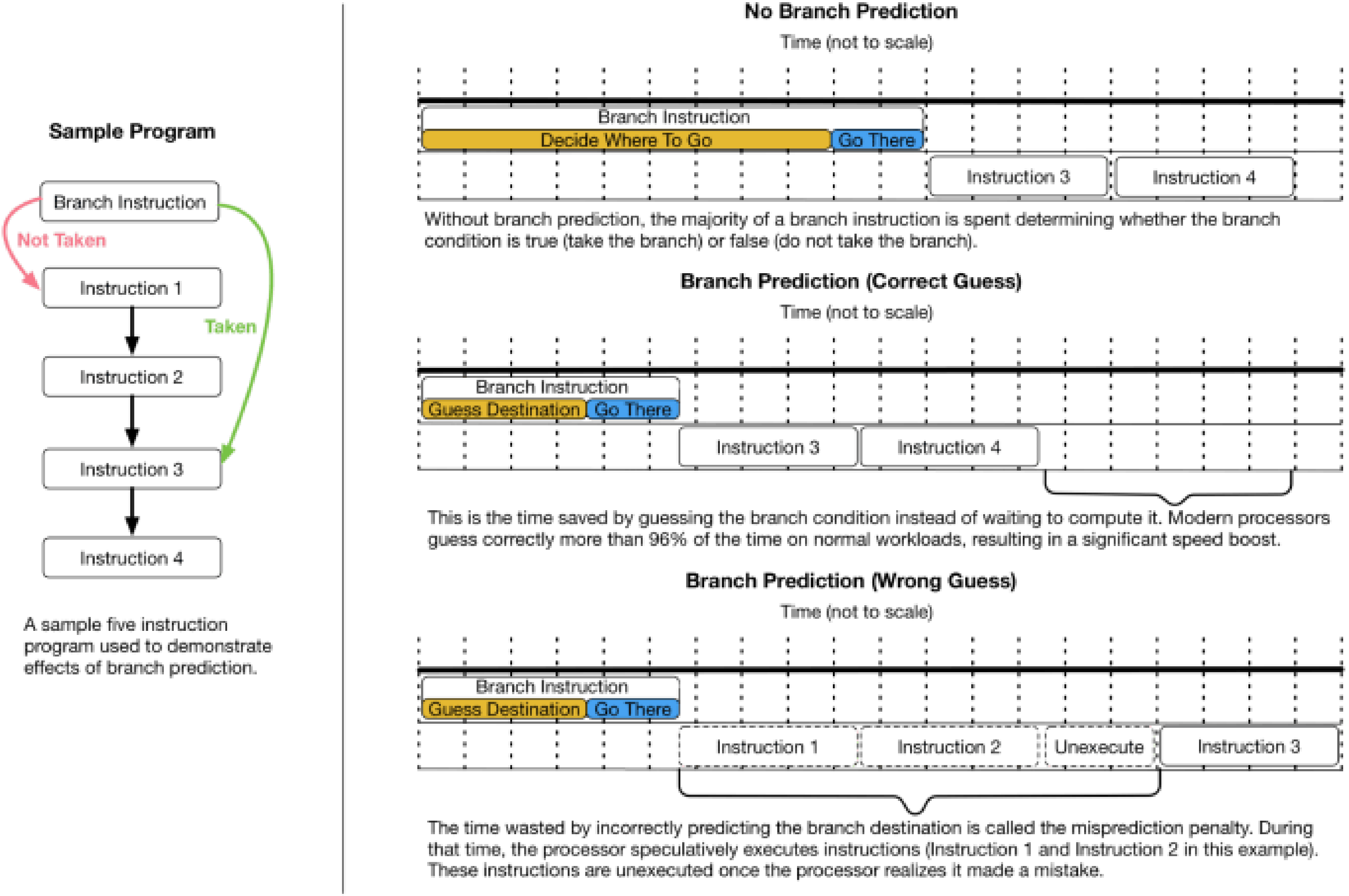
3. Stop leaving OS memory in the page table ✔
   - Would slow all computers down somewhat
   - Kernel Page Table Isolation
     - Estimated 5-30% performance loss
     - Improved by use of PCID bit in TLB

Spectre

- Speculative execution targeting branch prediction

- Disclosed in January 2018 by
  - Jann Horn (Google Project Zero) and
  - Paul Kocher in collaboration with, in alphabetical order, Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61)

Recall: Branch Prediction



Incredibly accurate in modern day computers >95%

Spectre v1

- Repeat meltdown-style attack using conditional branches
  - Conditional branches are especially prevalent for bounds checks in software virtual machines (like Javascript runtime)

1. Train conditional branch predictor that bounds check branch always succeeds
2. Make an invalid bounds-checked read, affecting cache state
3. Use cache timing analysis to determine value of read byte

Spectre v2

- Combine indirect branch prediction and in-kernel ROP gadgets
  - Indirect branch predictors try loading a guessed address

1. Train indirect branch predictor to go to a particular address
2. Make a system call requesting something
3. Within the system call, a branch mis-prediction that runs the targeted gadget, affecting cache state
   - Note: the gadget runs with kernel permission on physical memory
4. Use cache timing attack to determine result

Ramifications of speculative execution attacks

- Particularly big deals in the era of cloud computing
  - Anyone can run a program on an AWS server
  - And now can maybe read data from the other running programs…

- Speculative execution attacks are a new era for computer security
  - Hardware is still being actively developed to address attacks
    - Websites can be fixed in hours, Programs in days, OSes in weeks, and Hardware takes years
  - Attacks are still being developed
  - OS continues to have to adapt to both sides

Outline

- Design for security
- Memory attacks and defenses
  - Buffer overflow and No-Execute bit
  - Return-Oriented Programming and Address Space Layout Randomization
- Speculative execution attacks
  - Meltdown
  - Spectre