


# Midterm Review and Introduction to PA3: Side Channels

Adapted from slides by Ariana Mirian, Stefan  
Savage, Kevin Yu, and CSE 127 Sp 2020

A dark blue diagonal gradient bar that starts from the bottom left corner and extends towards the top right corner, covering the lower half of the slide.

# Agenda

## Midterm Review

- Tomorrow on Canvas between 8am and 10pm PST--start by 9pm!
- No questions: tech support only
- YES: all course content up to crypto
- YES: all readings up to crypto
- NO: “what is on the midterm?”
  - Anything may be

## PA3

- Due 5/6/21
- (PA4 will overlap slightly,, so plan ahead!)
- Side channel attacks
- Two parts:
  - Cache attacks
  - Timing attacks

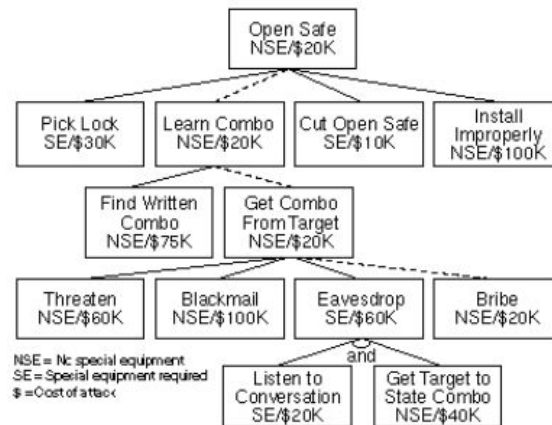
# Review: Threat Modeling & Risk Assessment

## First question: What is your threat model?

- What do you want to protect from whom?
- Who/what do you trust?
- Defines the scope of the problem

## Risk Assessment (ideal)

1. Start by understanding system requirements
2. Identify assets and attackers
3. Establish security requirements
4. Evaluate system design
5. Identify threats and classify risks
6. Address identified risk



[https://www.schneier.com/academic/archives/1999/12/attack\\_trees.html](https://www.schneier.com/academic/archives/1999/12/attack_trees.html)

**Think about how to violate assumptions!**

# Review: Control Flow Vulnerabilities

## What is a buffer overflow?

- Look over PA2 & understand HOW the exploits work.
- What assumptions do buffer overflows violate?
- Where do buffer overflows typically occur (Python data analysis vs. system-level C) and why?
- What is the problem with `gets()` and `strcpy()`?

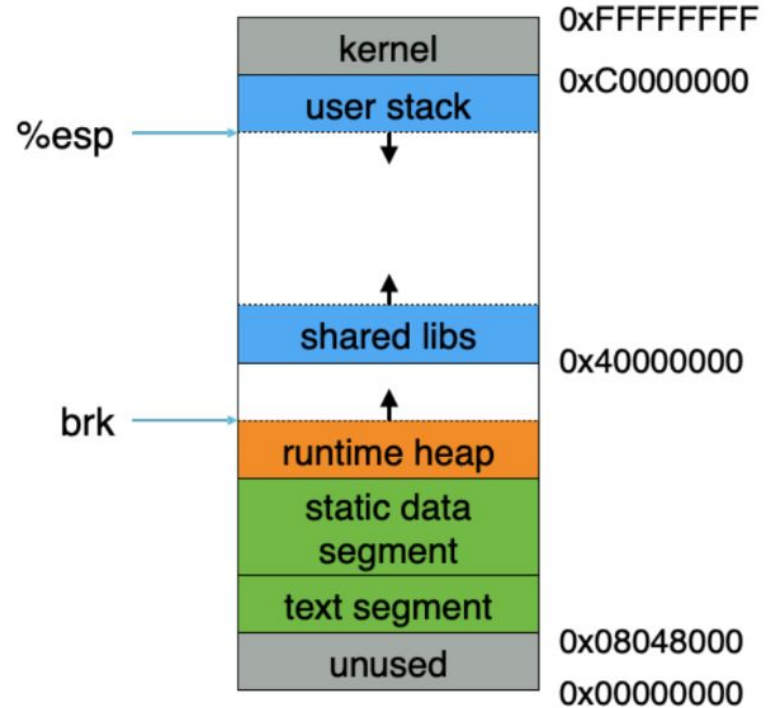
# The Stack

- Stack
  - Locals, call stack
- Heap
  - i.e. malloc, new, etc
- Data segments (globals, statics)
  - .data, .bss
- Text Segment (executable code)

**What does a stack frame look like?**

**What does a function call do?**

**What happens when a function call ends?**



# Review: Control Flow Vulnerabilities

**What is the ultimate goal of a buffer overflow attack, and what are the capabilities of the attacker?**

What are different ways to exploit a buffer overflow?

Format string vulnerability

Integer conversion: overflows, type casts

Heap vulnerability

What defenses exist to stop buffer overflows, and what attacks get around these defenses?



# Review: Control Flow Vulnerabilities

## What's the problem with printf()?

- Variadic function—variance in what can be input.
- What do the following vulnerabilities do?

```
printf("\x10\x01\x48\x08%x%x%x%x%s")
```

```
printf("%n")
```

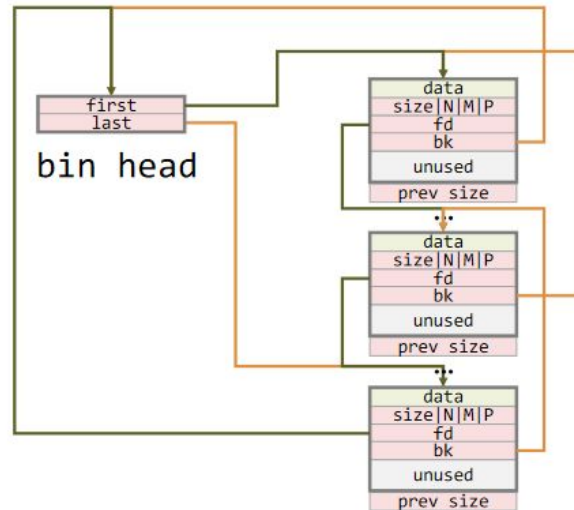
## What happens to freed memory in the heap?

Double-free

Use-after-free

- Unlink operation to remove a chunk from the free list:

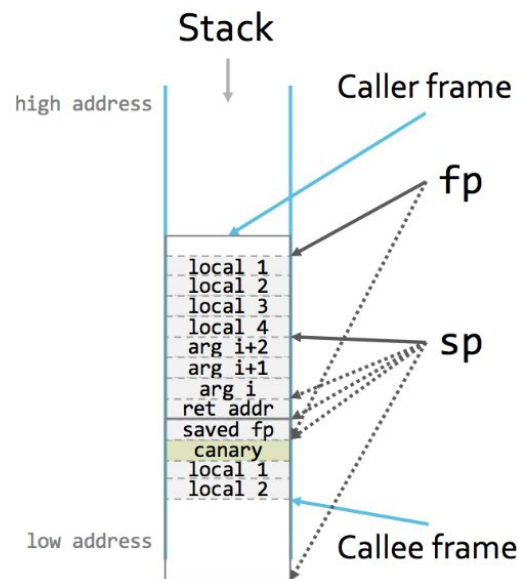
```
#define unlink(P, BK, FD)
{
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```



# Review: Control Flow Defenses

## Stack cookies/canaries

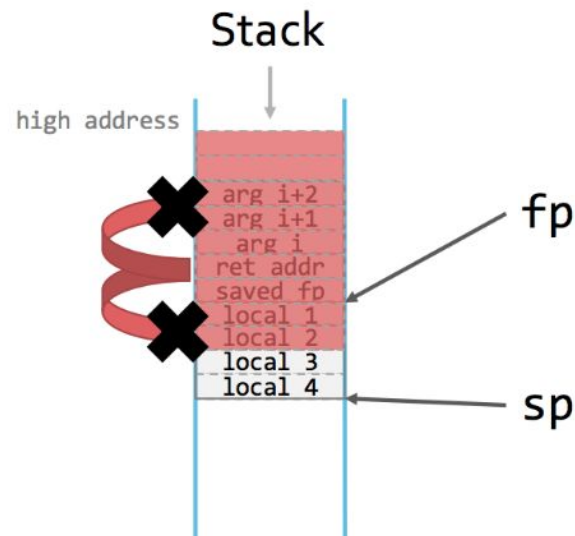
- Detect overwriting of return address
- Changes how callee works
  - Needs to allocate space for canary and push it (when calling)
  - Check canary (when returning)
- Can use fixed or random value or terminator canary
  - What are the tradeoffs of each?
- How do we bypass canaries in different scenarios?
- Are they still used today? (yes)



# Review: Control Flow Defenses

## Memory Protection (DEP, W^X)

- Make all pages writeable OR executable
  - Why does this help us?
- What are the tradeoffs?
  - Little performance impact vs required hardware support
  - A few others...
- How do we bypass this?
  - Tip: libc() and ROP

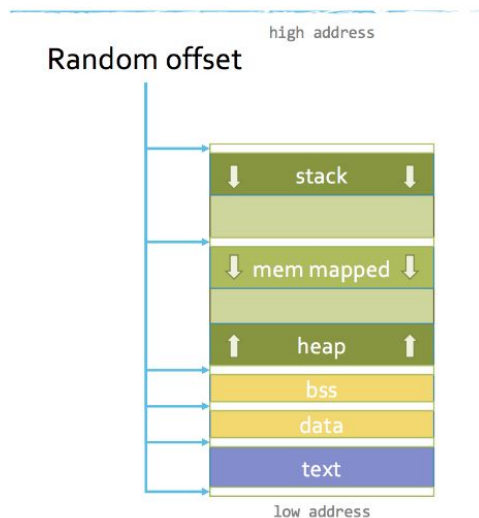


# Review: Control Flow Defenses

## Address Space

### Layout Randomization (ASLR)

- Randomize Stack base
  - Add random offset
  - Are addresses still relative?
- Bypasses?
  - How?
- Limitations?
  - Performance vs. protection



### Bypasses:

- NOP sleds
- Guessing!
- Leaking (e.g., with printf)
- Heap spray

# Review: Control Flow Integrity

## Recall:

Return-oriented programming chains “gadgets” from existing code into shellcode

Libc() usually enough to produce exploit

Can also use jumps/calls between gadgets.

Maybe start in middle of an instruction!

**Control-Flow Integrity** enforces that only *legitimate* jumps can take place.

What is legitimate?

- back to calling function
- expected in the source
- only to beginnings of functions.

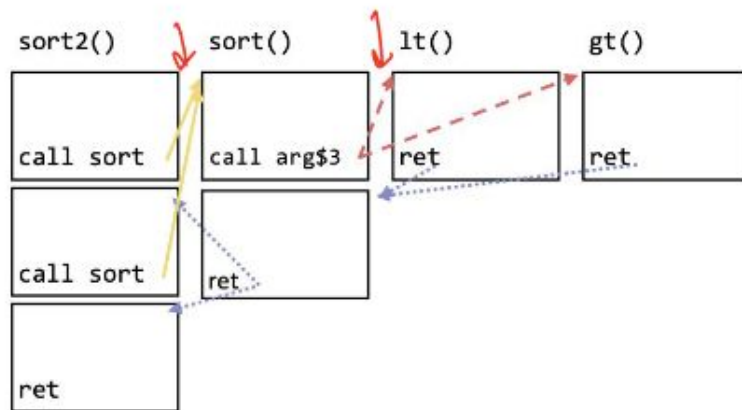
Needs to analyze source ahead of time.

# Review: Control Flow Integrity

```
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;  
}
```



# Review: Systems Security

## So why doesn't the entire system crash constantly?

### Secure design principles

- **Least privilege**
  - Only provide as much privilege to a program as is *needed* to do its job
- **Privilege separation**
  - Divide system into different pieces, each with separate privileges, requiring multiple different privileges to access sensitive data/code (AND vs OR)
- **Complete mediation**
  - Check **every** access that crosses a trust boundary against security policy
- **Defense in depth**
  - Use more than one security mechanism (belt and suspenders)
- **Simple designs are preferred**

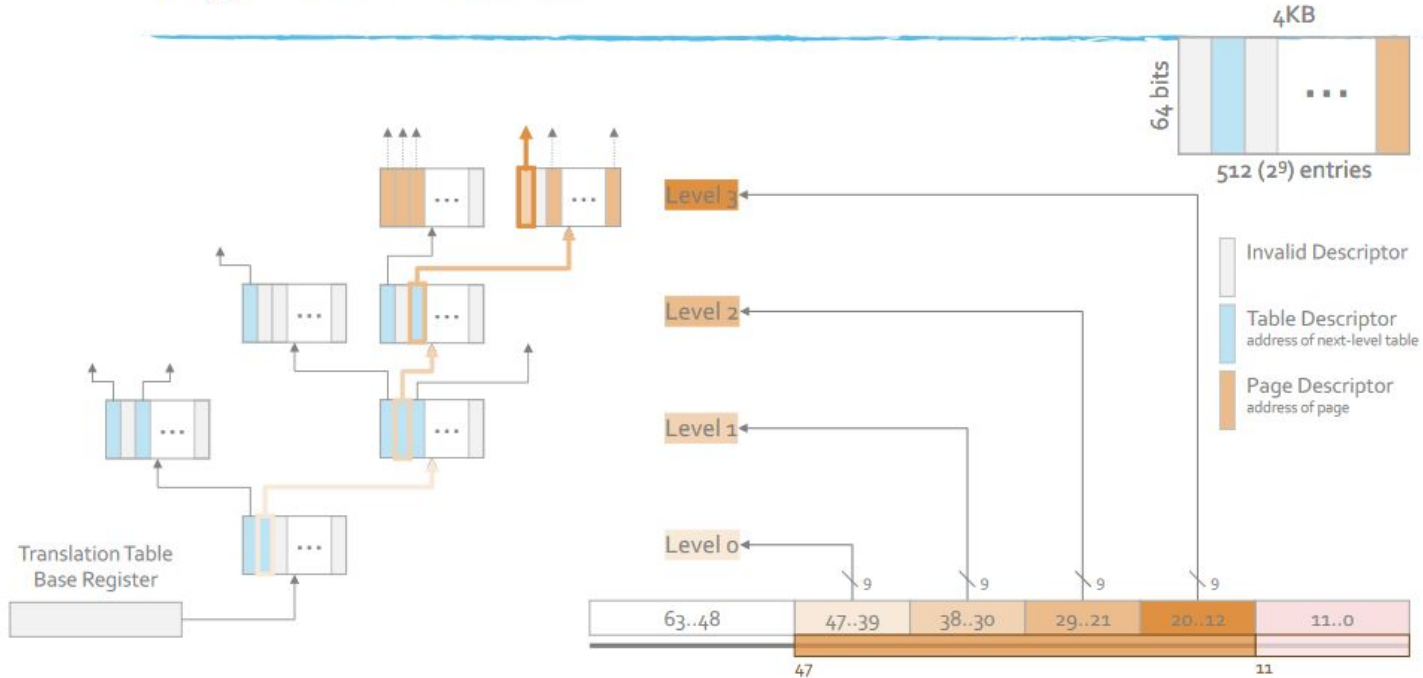
# Review: Systems Security

## How do operating systems implement these principles?

- Process abstraction & isolation
- User IDs & Access Control Lists
- Hardware support
- User/Kernel privilege separation
- Virtual Memory & Address Translation
- Page tables!
  - How do these work?
  - How do we make syscalls faster, and what can go wrong? (hint: return-to-user)

# Review: Systems Security

## Page Table Walk



# Review: Side Channels

How do we determine where trusted boundaries are, and what crosses those boundaries?

Intended forms of access + unintended forms of access: side channels

--because we must implement software

--what are some examples of side channels?

Understand prominent side-channel attacks:

Rowhammer (DRAM errors), Spectre, Meltdown (speculative execution sidechannels)



# PA3: Logistics

Due 5/6/2021, 11:00 AM PST on Gradescope

Download VM from Assignments page on course website.

Contains 3 folders: base, memhack, timehack.

DO THIS FIRST: make generate with your student ID.

Then: two attacks. Both use `sysapp.c`, which contains two functions `check_pass()` and `hack_system()`.

- 1) Memhack: use page faults to identify password
- 2) Timehack: use timing in password check to identify password.

Advice: your `check_pass` function takes a username and a password. Your `memhack.c` and `timehack.c` files will have a username chosen for you. Use this everywhere. We will use the same username for grading.

DO: repeat your experiments many times!

DON'T: try to hard-code or guess the password.

# PA3: sysapp.c

## ▶ check\_pass

- ▶ password to check (**\*pass**) is passed by reference
- ▶ **check\_pass** loops over characters checking against true password sequentially
- ▶ **correct\_pass** is static in the given vm, but its value will change for grading so solution should generalize
- ▶ **delay** is added to make time hack more feasible

## ▶ hack\_system

- ▶ solution should call this on the password when it is found

```
void delay() {
    int j, q;
    for (j = 0; j < 100; j++) {
        q = q + j;
    }
}

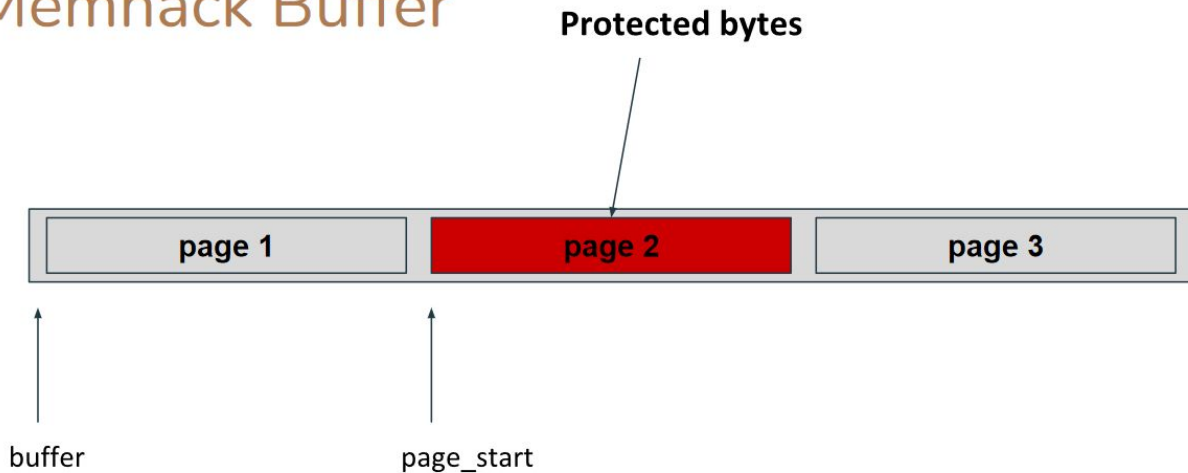
int check_pass(char *pass) {
    int i;
    for (i = 0; i <= strlen(correct_pass); i++) {
        delay(); // artificial delay added for timehack
        if (pass[i] != correct_pass[i])
            return 0;
    }
    return 1;
};

void hack_system(char *correct_pass) {
    if (check_pass(correct_pass)) {
        printf("OK: You have found correct password: '%s'\n", correct_pass);
        printf("OK: Congratulations!\n");
        exit(0);
    } else {
        printf("FAIL: The password is not correct! You have failed\n");
        exit(3);
    }
};
```

# PA3: memhack.c

The file is configured to give you a buffer that runs into unmapped memory.

## Memhack Buffer



Use `sigsetjmp/siglongjmp` to detect page fault errors: `demonstrate_signals` is an example of this.

# PA3: timehack.c

## Same sysapp.c, different side channel

How long does it take to check a password if the first 5 characters are correct? 10?

rdtsc() gives a number that increases with each cycle.

- Avoid using printf's in your code, as they can cause huge variances in execution time
- Use the median and not the mean for the multiple trials you run for a given guess
  - Outliers are every extreme, so you want to avoid using the mean
  - qsort may be helpful
- If time is not continuing to increase as you progress through characters, then you probably made a bad guess earlier. Backtrack