CSE 484/M584: Computer Security (and Privacy)

Spring 2025

David Kohlbrenner
dkohlbre@cs

UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner, Nirvan Tyagi. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials

Admin

- Office hours start today!
- Ed board is open
  - Our target is 24hrs for replies
  - Spend time reading/looking up resources before asking questions
- Lab 1 is out
  - Lab 1a (exploit 1+2) are due Wednesday night.
  - See Gradescope for the handins.
  - Reminder about policies
- 584 students: you have a reading due tonight!

Threat Modeling: Again

Gradescope!

- As in, lets *threat model part of Gradescope* todays in-class activity

Gradescope! – Gradescope Group handins

- How do group handins on Gradescope work?
- Who might be an adversary that would abuse this system?
- What might their goal be?
- What might an asset be?
- *How should we think about defense against this threat?*

Thinking about Defense

Approaches to Defense

- Prevention
  - Stop an attack
- Detection
  - Detect an ongoing or past attack
- Response and Resilience
  - Respond to / recover from attacks

- The threat of a response may be enough to deter some attackers

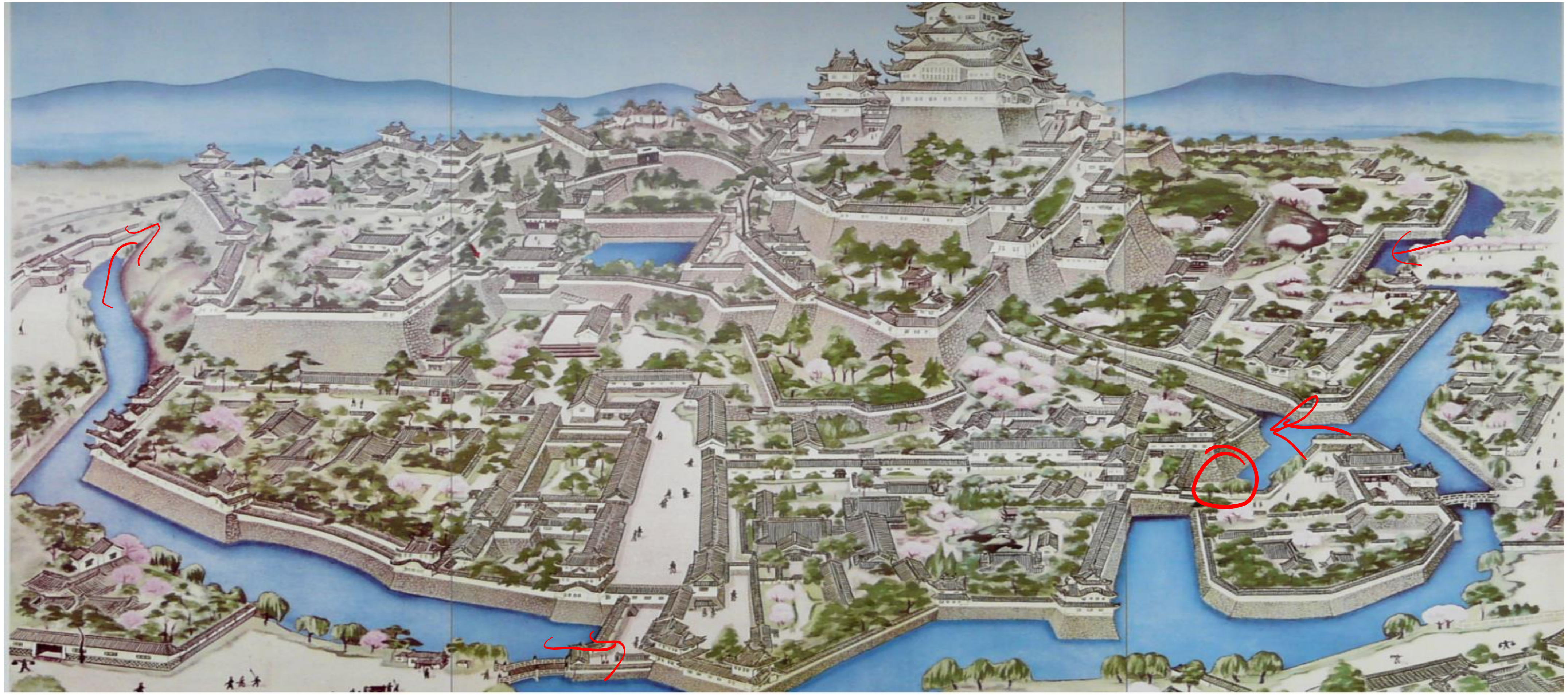
Whole System is Critical

- Securing a system involves a whole-system view
  - Cryptography
  - Implementation
  - People
  - Physical security
  - Everything in between

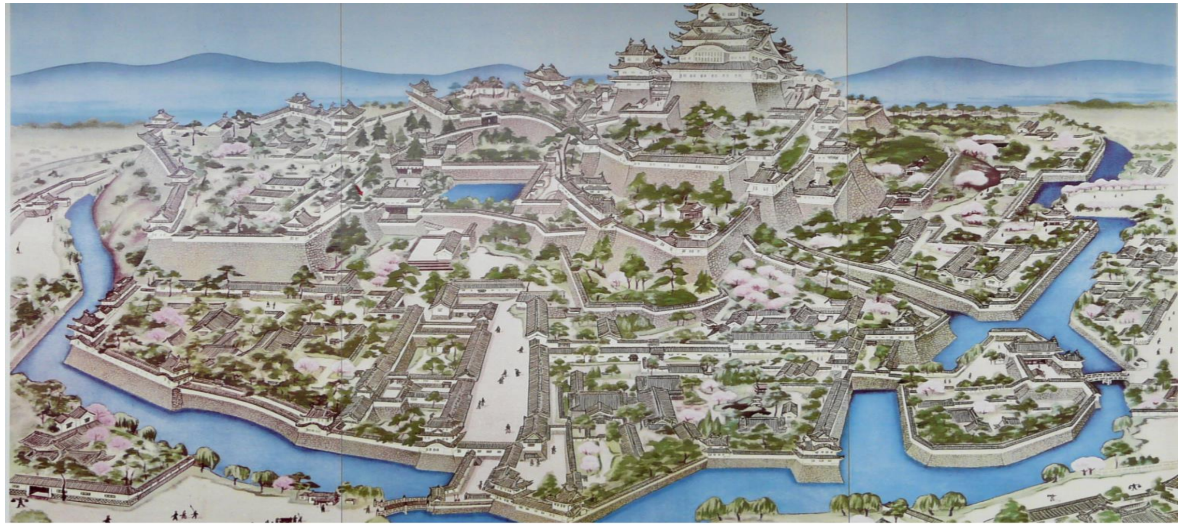
- This is because "security is only as strong as the weakest link," and security can fail in many places
  - No reason to attack the strongest part of a system if you can walk right around it.

Asymmetric advantages in security

Asymmetric advantages in security

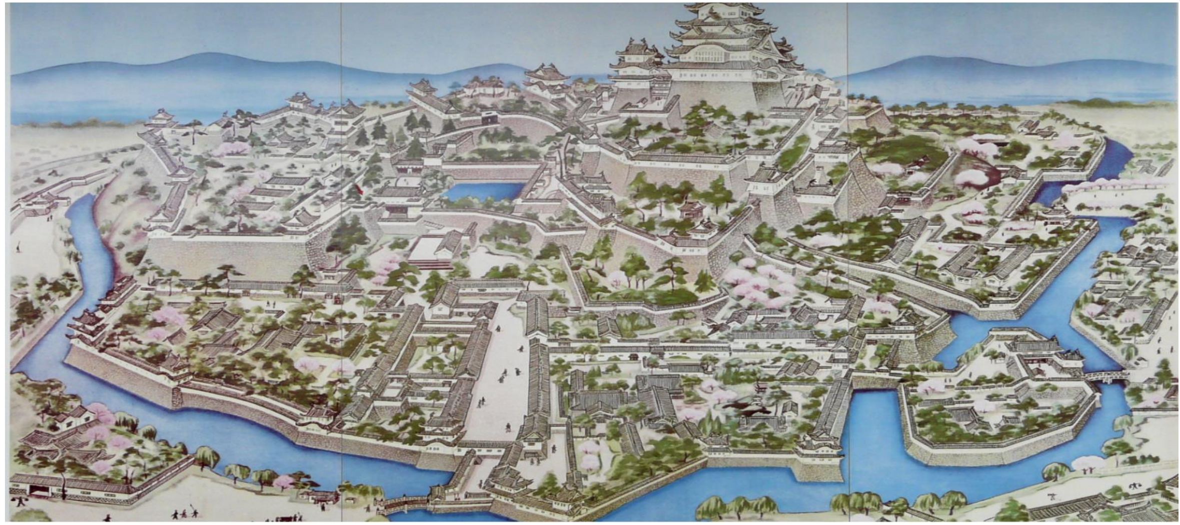


Attacker’s Asymmetric Advantage



- Attacker only needs to win one time, not all the time
- Attackers are professional attackers (maybe)

Defender’s Asymmetric Advantage



- The attacker only succeeds while undetected
- Defender is on ‘home turf’
- Defender has (hopefully) more resources than the attacker
- If the defender can spot them one time, they win

Podcast: https://risky.biz/category/risky-business-news/

Better News

- There are a lot of defense mechanisms
  - We’ll study some, but by no means all, in this course
- It’s important to understand their limitations
  - “If you think cryptography will solve your problem, then you don’t understand cryptography… and you don’t understand your problem” -- Bruce Schneier (… definitely not Bruce)

Binary Exploitation: Continued

A note on assembly

4B

objdump

- Its all x86_32 assembly for Lab 1
- There are two syntaxes (I'm sorry)
  - AT&T (default on Linux, GAS)
  - Intel (easier to read, IMO, default(?) in gef)

mov ebx, ecx

int b = c;

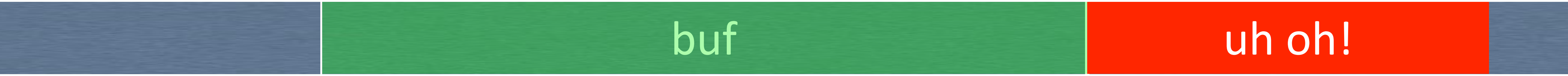
% $

mov ebx, ecx

Attacks on Memory Buffers

- Buffer is a pre-defined data storage area inside computer memory (stack or heap)
- Typical situation:
  - A function takes some input that it writes into a pre-allocated buffer.
  - The developer forgets to check that the size of the input isn’t larger than the size of the buffer.
  - Uh oh.
    - “Normal” bad input: crash
    - “Adversarial” bad input : take control of execution

Stack Buffers



buf

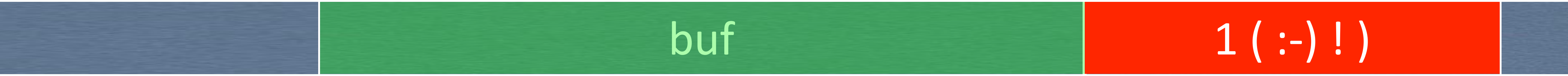
uh oh!

- Suppose Web server contains this function

```
void func(char *str) {
   char buf[126];
   ...
   strcpy(buf,str);
   ...
}
```

- No bounds checking on strcpy()
- If str is longer than 126 bytes
  - Program may crash
  - Attacker may change program behavior

Example: Changing Flags



buf

1 (:-) !)

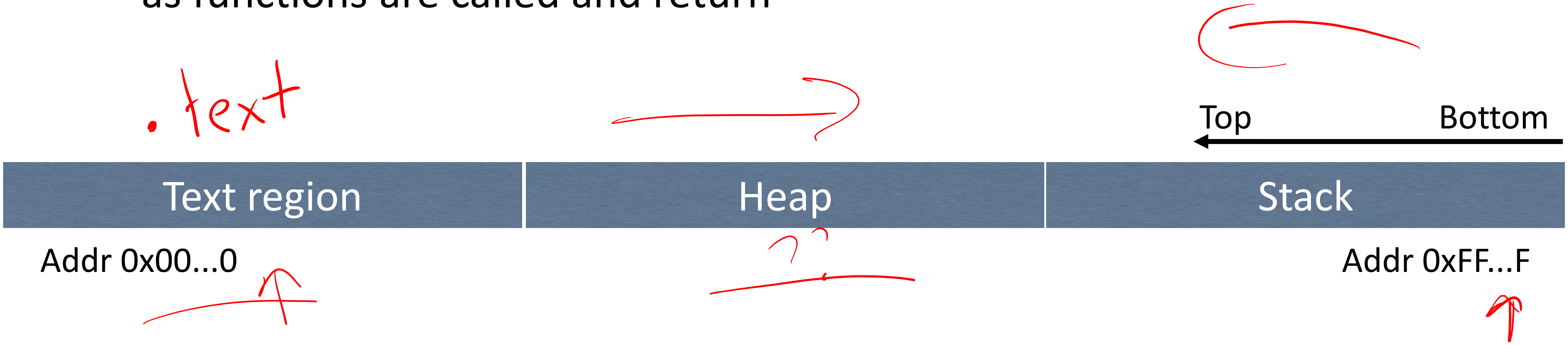
- Suppose Web server contains this function

```
void func(char *str) {
      byte auth = 0;
   char buf[126];
   ...
   strcpy(buf,str);
   ...
}
```

- Authenticated variable non-zero when user has extra privileges
- Morris worm also overflowed a buffer to overwrite an authenticated flag in fingerd

Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return



Stack Buffers

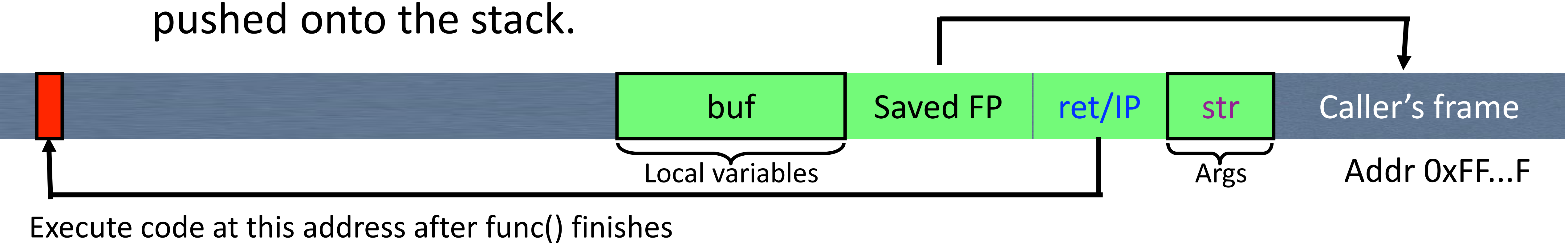
- Suppose Web server contains this function:

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
}
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame (activation record) is pushed onto the stack.



Execute code at this address after func() finishes

What if Buffer is Overstuffed?

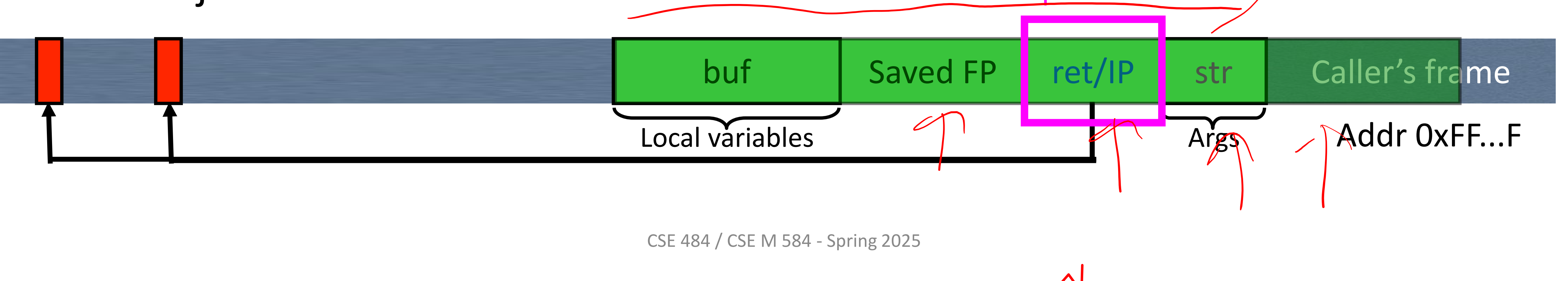
- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
}
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

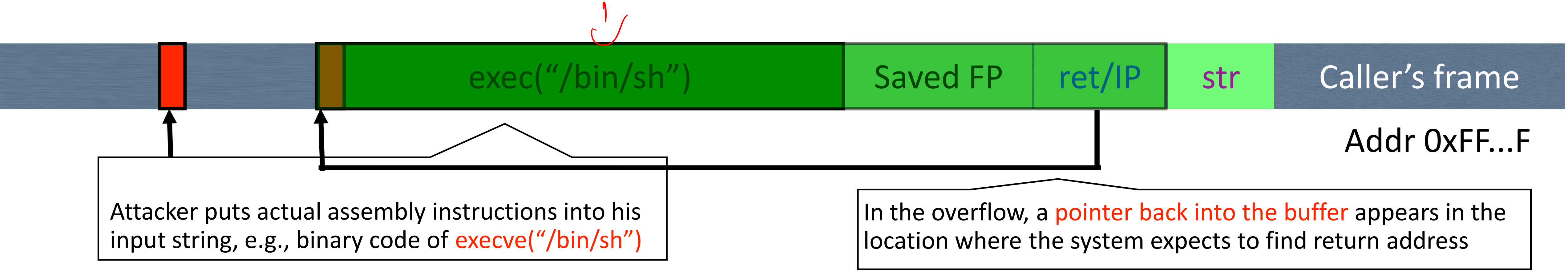
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

This will be interpreted as return address!



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



Attacker puts actual assembly instructions into his input string, e.g., binary code of execve("/bin/sh")

In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address

- When function exits, code in the buffer will be executed, giving attacker a shell **("shellcode")**
  - Root shell if the victim program is setuid root

Buffer Overflows Can Be Tricky to exploit…

- The input string must write the correct address of attack code in the saved return address
  - The value overwriting the saved return address must point to executable code
    - Otherwise application will (probably) crash with segfault

- Attacker must also correctly store executable code somewhere…
  - And then know the address of that code!

0xFFAACCDD

Classic problem: Lack of bounds checks

- `strcpy(buf, str)`
  - strcpy does not check input size
  - simply copies memory contents into buf starting from *str until “\0” (NUL/NULL byte) is encountered, ignoring the size of area allocated to buf

0x00

- Many C library functions are unsafe in this way!
  - `strcpy(char *dest, const char *src)`
  - `strcat(char *dest, const char *src)`
  - `gets(char *s)`

0x00

- Or other interesting ways
  - `scanf(const char *format, …)`
  - `printf(const char *format, …)`

“0x00”

When Does Bounds Checking Help?

- `strncpy(char *dest, const char *src, size_t n)`
  - Limits copy length to whatever ‘n’ is

“user”“:”“cpw”

- Potential overflow in htpasswd.c (Apache 1.3):

```
strcpy(record, user);
strcat(record, ”:”);
strcat(record, cpw);
```

Copies username (“user”) into buffer (“record”), then appends “:” and hashed password (“cpw”)

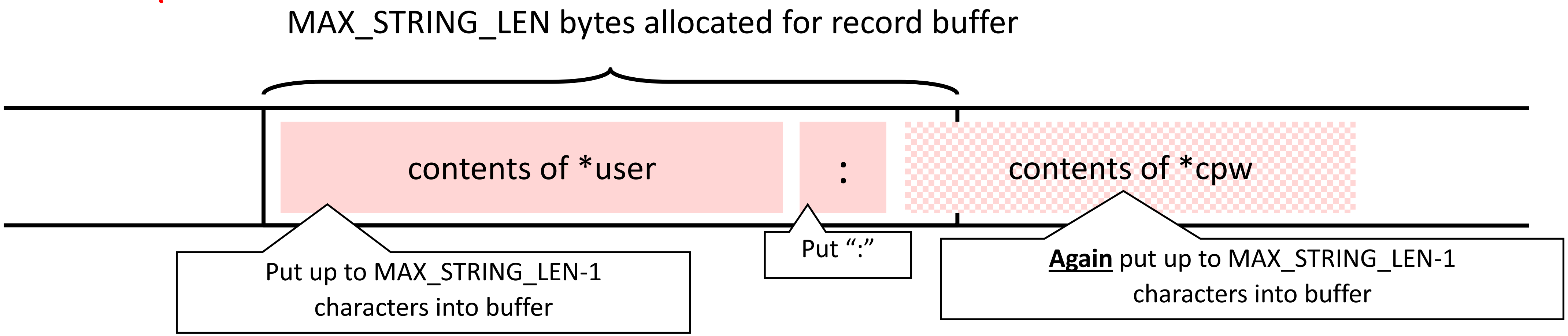
- Published fix:

```
strncpy(record, user, MAX_STRING_LEN-1);
strcat(record, ”:”);
strncat(record, cpw, MAX_STRING_LEN-1);
```

Misuse of strncpy in htpasswd "Fix"

- Published "fix" for Apache htpasswd overflow:

```
strncpy(record, user, MAX_STRING_LEN-1);
strcat(record, ":");
strncat(record, cpw, MAX_STRING_LEN-1);
```



MAX_STRING_LEN bytes allocated for record buffer
contents of *user
:
contents of *cpw
Put up to MAX_STRING_LEN-1 characters into buffer
Put ":"
Again put up to MAX_STRING_LEN-1 characters into buffer

What About This? – Homebrew copy?

```
void mycopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
    }  "\0"

void main(int argc, char *argv[]) {
    if (argc==2)
        mycopy(argv[1]);
    }
```

What About This? – Homebrew copy?

```
void mycopy(char *input) {
    char buffer[512]; int i;
    for (i=0; i<=512; i++)
        buffer[i] = input[i];
    }
void main(int argc, char *argv[]) {
    if (argc==2)
        mycopy(argv[1]);
    }
```

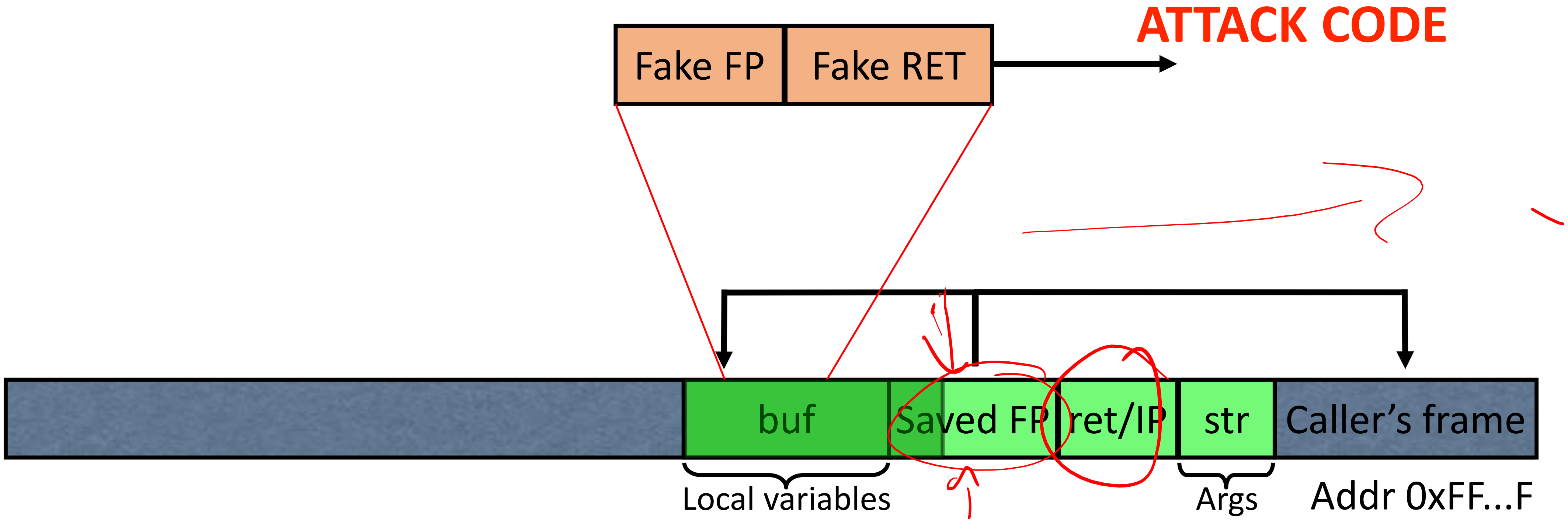
This will copy **513** characters into buffer. Oops!

exploit 2

1-byte overflow: can’t change RET, but can change pointer to previous stack frame…

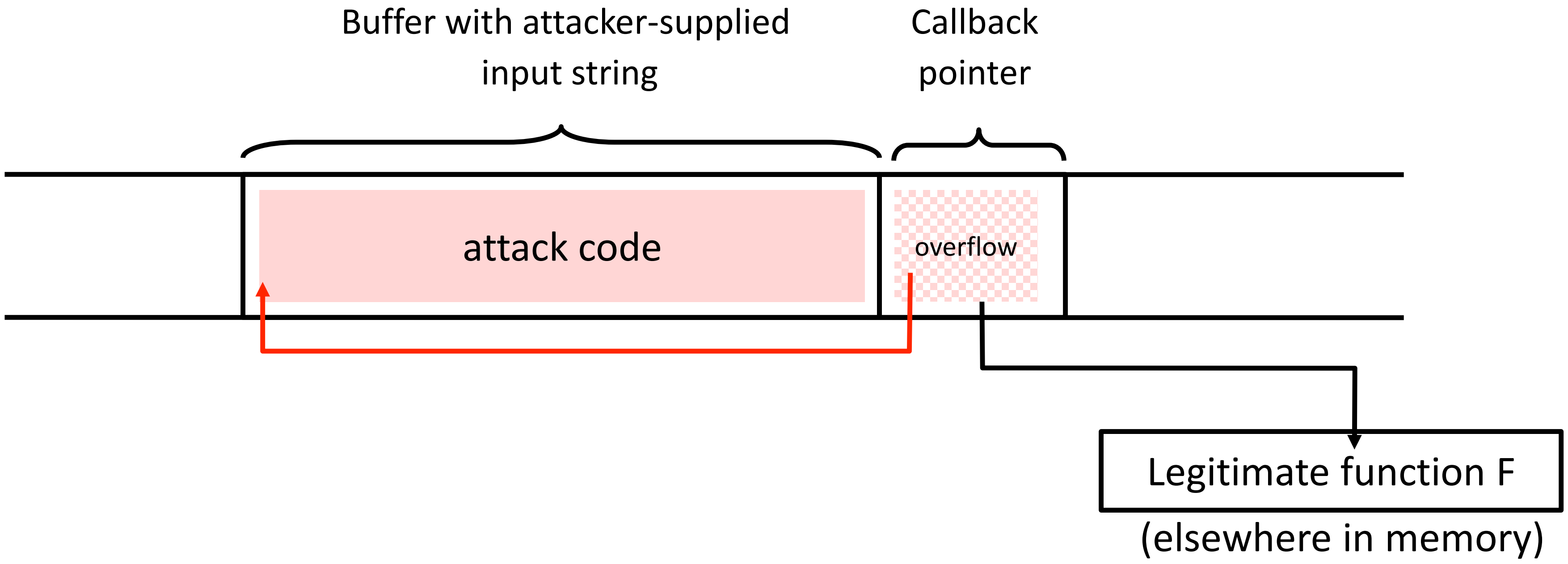
Frame pointers (and saved frame pointers)

Frame Pointer Overflow



Another Variant: Function Pointer Overflow

- C uses function pointers for callbacks: if pointer to F is stored in memory location P, then one can call F as (*P)(…)


Buffer with attacker-supplied input string
Callback pointer
attack code
overflow
Legitimate function F
(elsewhere in memory)