

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



[Home](#) [News](#) [Sport](#) [Business](#) [Innovation](#) [Culture](#) [Arts](#) [Travel](#) [Earth](#) [Audio](#) [Video](#) [Live](#)

Microsoft rolls out AI screenshot tool dubbed 'privacy nightmare'

2 hours ago

Share  Save 



Imran Rahman-Jones

Technology reporter

Admin





- Lab 1b due next Wednesday.
 - Start early (hopefully already)
- Homework 1 is posted
 - Should be relatively quick
 - Good chance to discuss something in detail with classmates!
- Reminders:
 - There are helpful exercises in your lab 1 repo!
 - There is a separate writeup on the assignments page for frame pointers and printf exploitation (no new information)

Threat modeling again again again

- You are taking a course that has a required assignment every lecture.
- The course uses Gradescope to manage those assignments, which go 'live' near the beginning of class, and are due at the end.
- How can you ensure you get credit for these assignments without attending class? 
- How might that approach be mitigated or detected? 



In-class components

- Please don't make this adversarial!
- If you come to class, complete the component during the designated time. 
- If you miss class, complete it *while watching the recorded lecture*.
 - Don't fill it out, and then watch lecture. 
-  • This type of think-pair-share and discuss format has well-studied benefits to learning
 - It also is very low stakes, and takes minimal amounts of time. 

Hardening binaries

Buffer Overflow: Causes and Cures

- Classical memory exploit involves **code injection**
 - Put malicious code at a predictable location in memory, usually masquerading as data
 - Trick vulnerable program into passing control to it
- Possible defenses:
 1. Prevent execution of untrusted code
 2. Detect overflows
 3. Validate pointers
 4. Address space layout randomization
 5. Code analysis
 6. Better interfaces
 7. ...

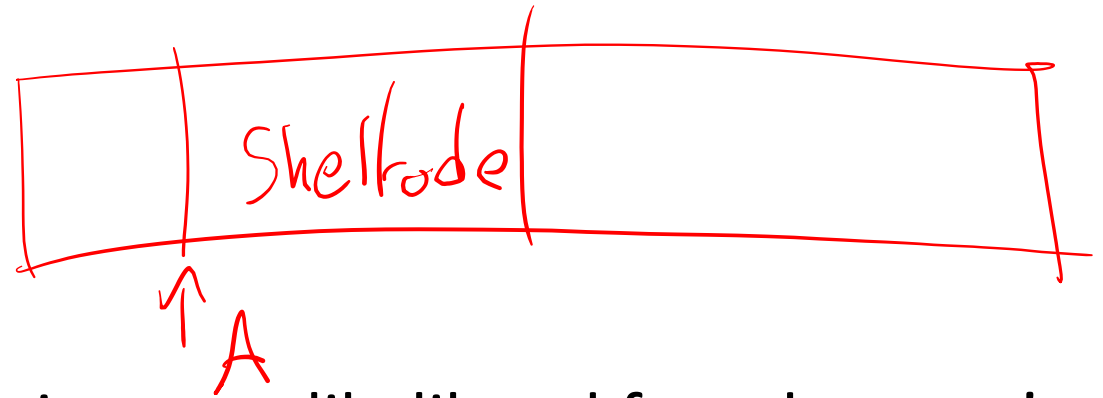
ASLR: Address Space Randomization

- Randomly arrange address space of key data areas for a process
 - Base of executable region
 - Position of stack
 - Position of heap
 - Position of libraries
- Introduced by Linux PaX project in 2001
- Adopted by OpenBSD in 2003
- Adopted by Linux in 2005

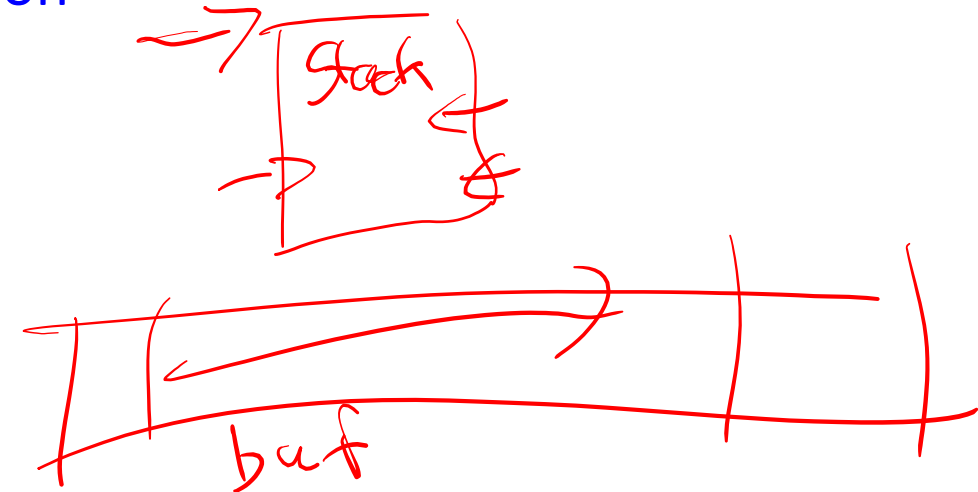
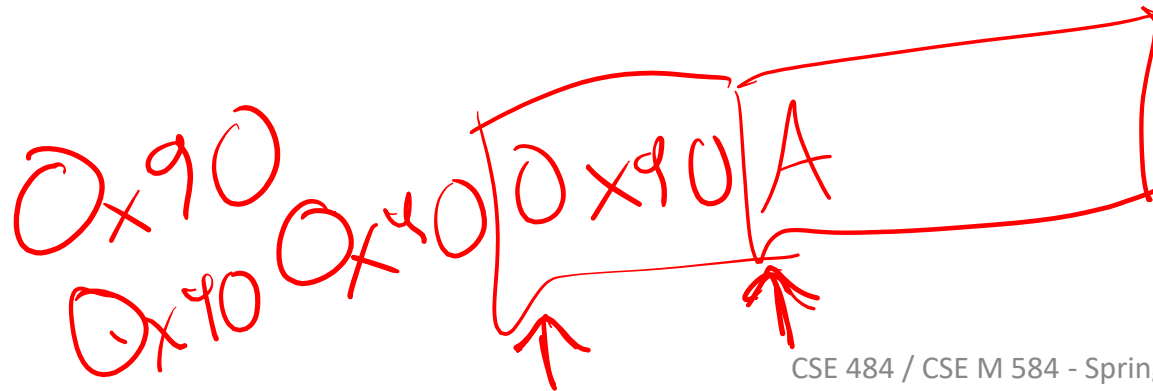
ASLR: Address Space Randomization

- Deployment (examples)
 - Linux kernel since 2.6.12 (2005+)
 - Android 4.0+
 - iOS 4.3+ ; OS X 10.5+
 - Microsoft since Windows Vista (2007)
- Attacker goal: Guess or figure out target address (or addresses)
 - (Think about how poor printf usage might help an attacker!)

Attacking ASLR



- **NOP sleds and heap spraying** to increase likelihood for adversary's code to be reached (e.g., on heap)
- **Brute force attacks or memory disclosures** to map out memory on the fly
 - Disclosing a single address can reveal the location of all code within a library, depending on the ASLR implementation
 - Remember our printf vulnerabilities!



Defense: Executable Space Protection

- Mark all writeable memory locations as non-executable
 - **This blocks many code injection exploits**
- Hardware support
 - AMD “NX” bit (no-execute), Intel “XD” bit (execute disable) (in post-2004 CPUs)
 - Makes memory page non-executable
- Widely deployed
 - Windows XP SP2+ (2004), Linux since 2004 (check distribution), OS X 10.5+ (10.4 for stack but not heap), Android 2.3+

What Does “Executable Space Protection” Not Prevent?

- Can still corrupt stack ...
 - ... or function pointers
 - ... or critical data on the heap
- **As long as RET points into existing code, executable space protection will not block control transfer!**
 - return-to-libc exploits

return-to-libc

- Overwrite saved return address with address of any library routine
- Does not look like a huge threat?
 - ...

return-to-libc

- Overwrite saved return address with address of any library routine
 - Arrange stack to look like arguments
- Does not look like a huge threat
 - ...
 - We can call *any* function we want!
 - Say, exec 😊

return-to-libc++

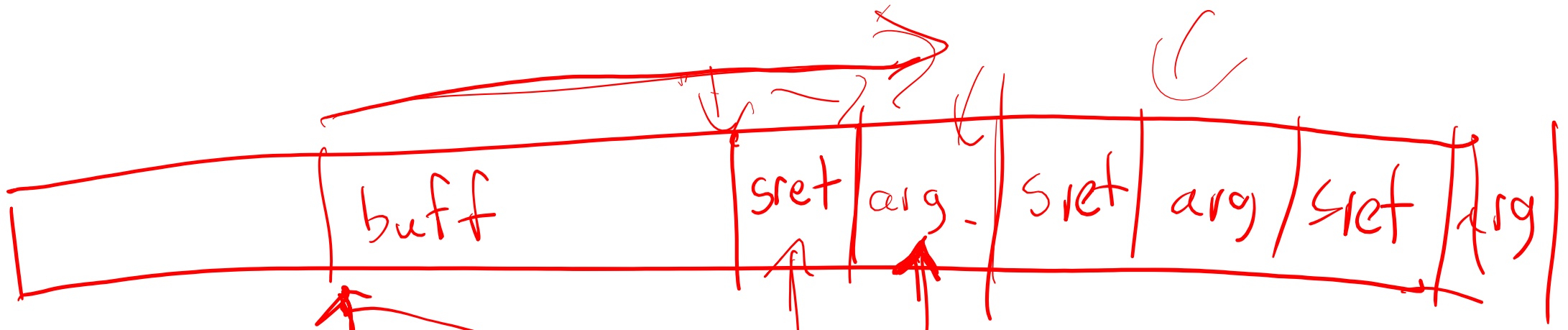
- Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
- **Any** existing instruction in the code image is fine
 - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
 - Execution will be transferred... to where?
 - Read the word pointed to by stack pointer (SP)
 - Guess what? Its value is under attacker's control!
 - Use it as the new value for IP
 - Now control is transferred to an address of attacker's choice!
 - Increment SP to point to the next word on the stack

Chaining RETs

- Can chain together sequences ending in RET
 - Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)
- What is this good for?
- Answer [Shacham et al.]: **everything**
 - Turing-complete language
 - Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
 - Attack can perform arbitrary computation using no injected code at all – **return-oriented programming**
- Truly, a “weird machine”

Return-to-libc

-> exec("/bin/sh");
↑
push eax
- call exec



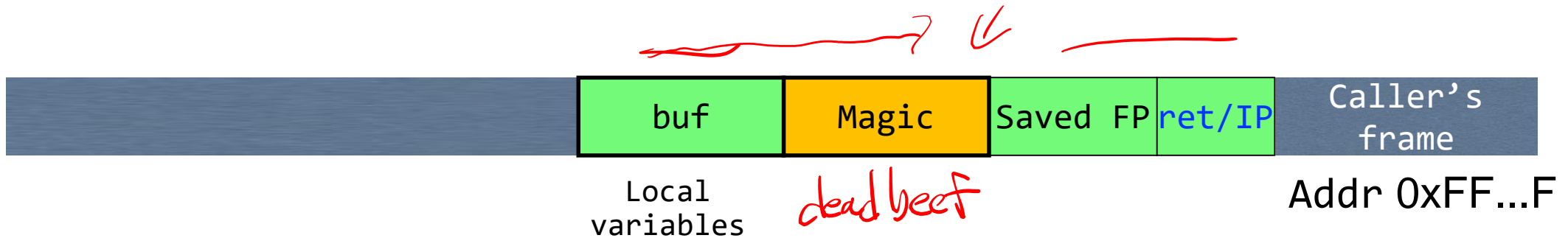
exec

-> add esp
ret

Defense: ???

glibcScope

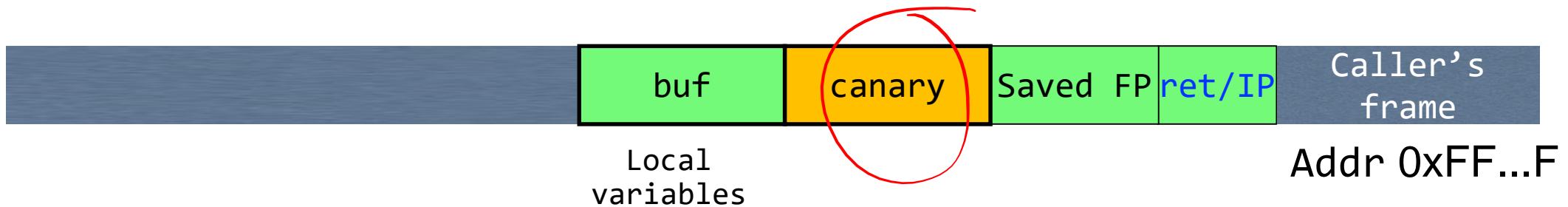
- • Choose a random value (Magic) at program startup
- Push that value onto the stack at the start of every function.



- • Now what?
- • If your adversary wants to do a classical stack-based buffer overflow, what will happen?
- How can we use this magic value for defense?

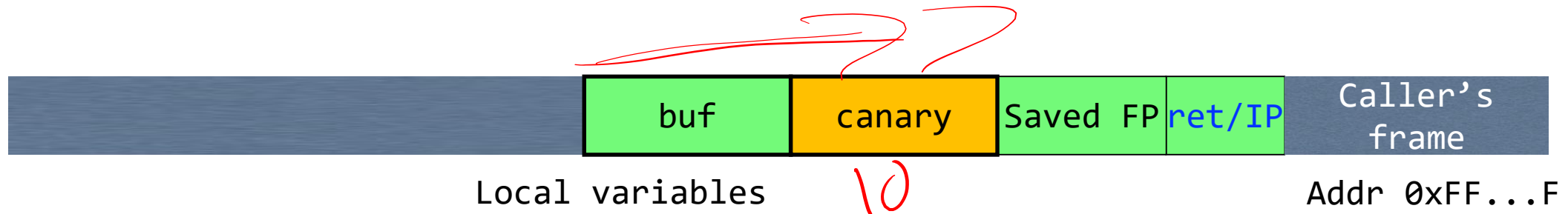
Defense: Run-Time Checking: StackGuard

- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
 - Any overflow of local variables will damage the canary



Defense: Run-Time Checking: StackGuard


- Embed “canaries” (stack cookies) in stack frames and verify their integrity prior to function return
 - Any overflow of local variables will damage the canary



- Choose random canary string on program start
 - Attacker can't guess what the value of canary will be
- Canary contains: “\0”, newline, linefeed, EOF
 - String functions like strcpy won't copy beyond “\0”



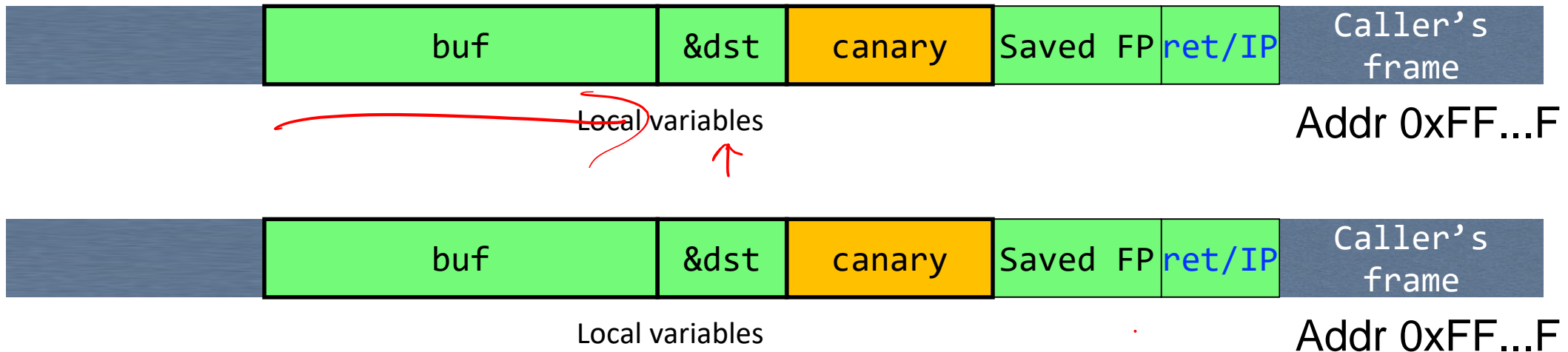
StackGuard Implementation

- StackGuard requires code recompilation 
- Checking canary integrity prior to every function return causes a performance penalty
 - For example, 8% for Apache Web server at one point in time

Defeating StackGuard

- StackGuard can be defeated
 - A single memory write where the attacker controls both the value and the destination is sufficient

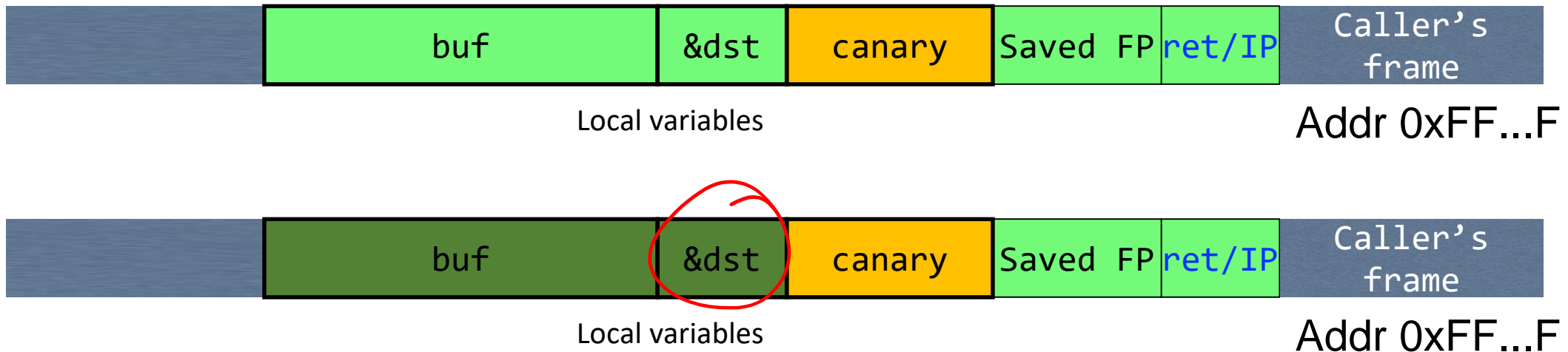
```
foo() {  
    char *dst;  
    char buf[...];  
    ...  
    strcpy(buf, readUntrustedInput());  
    strcpy(dst, buf);  
}
```



Defeating StackGuard

- StackGuard can be defeated
 - A single memory write where the attacker controls both the value and the destination is sufficient

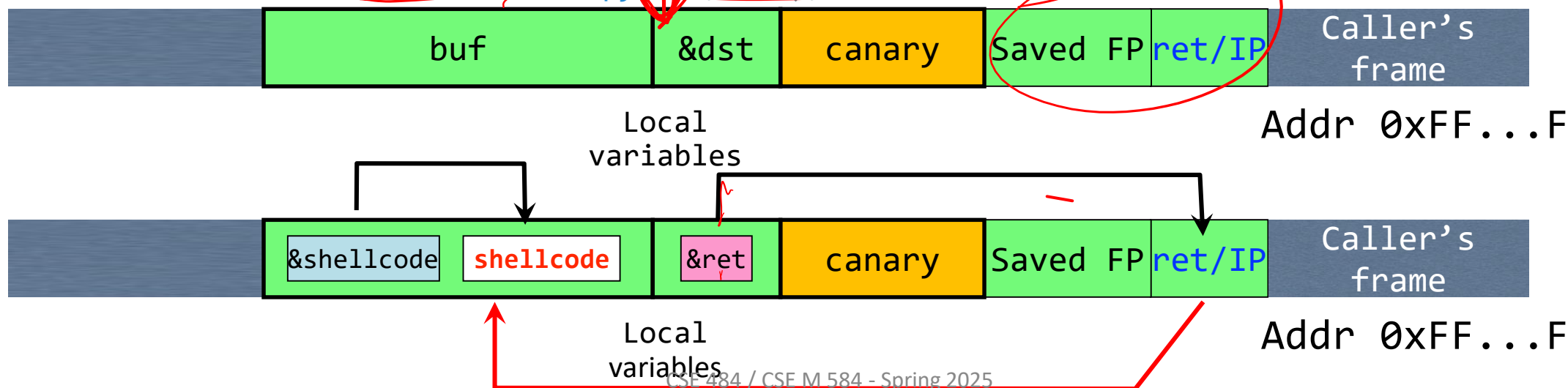
```
foo() {  
    char *dst;  
    char buf[...];  
    ...  
    strcpy(buf, readUntrustedInput());  
    strcpy(dst, buf);  
}
```



Defeating StackGuard

- StackGuard can be defeated
 - A single memory write where the attacker controls both the value and the destination is sufficient

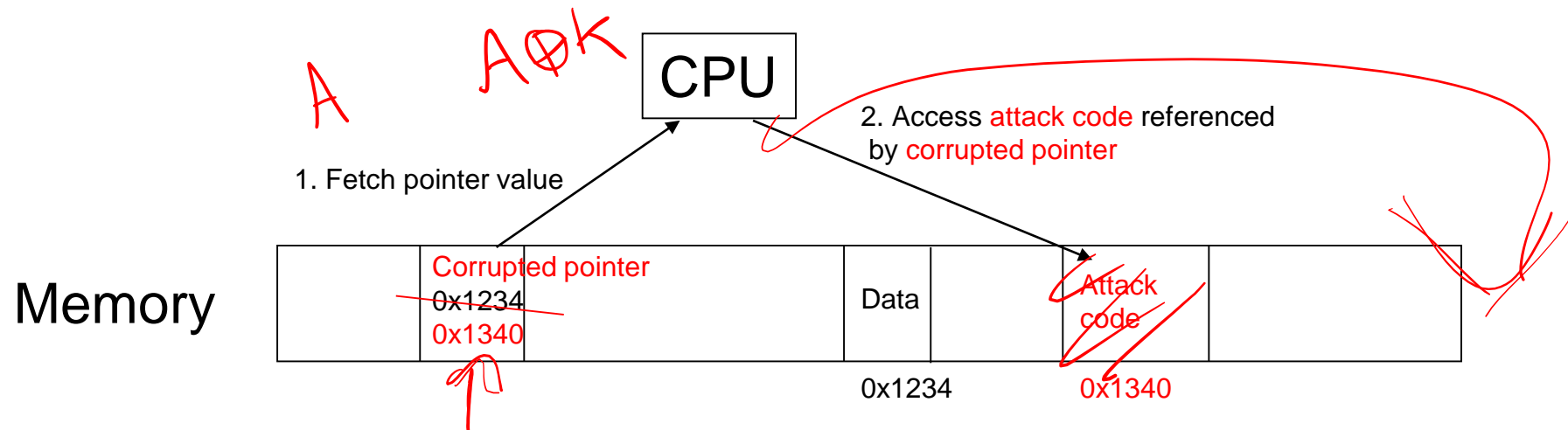
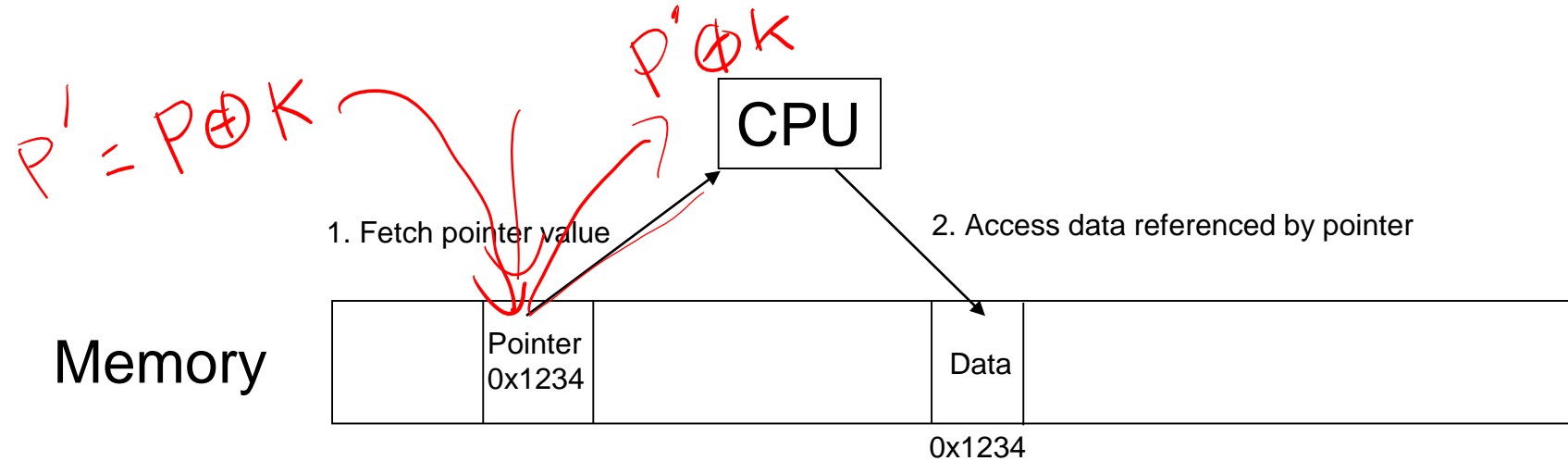
```
foo() {  
    char *dst;  
    char buf[...];  
    ...  
    strcpy(buf, readUntrustedInput());  
    strcpy(dst, buf);  
}
```



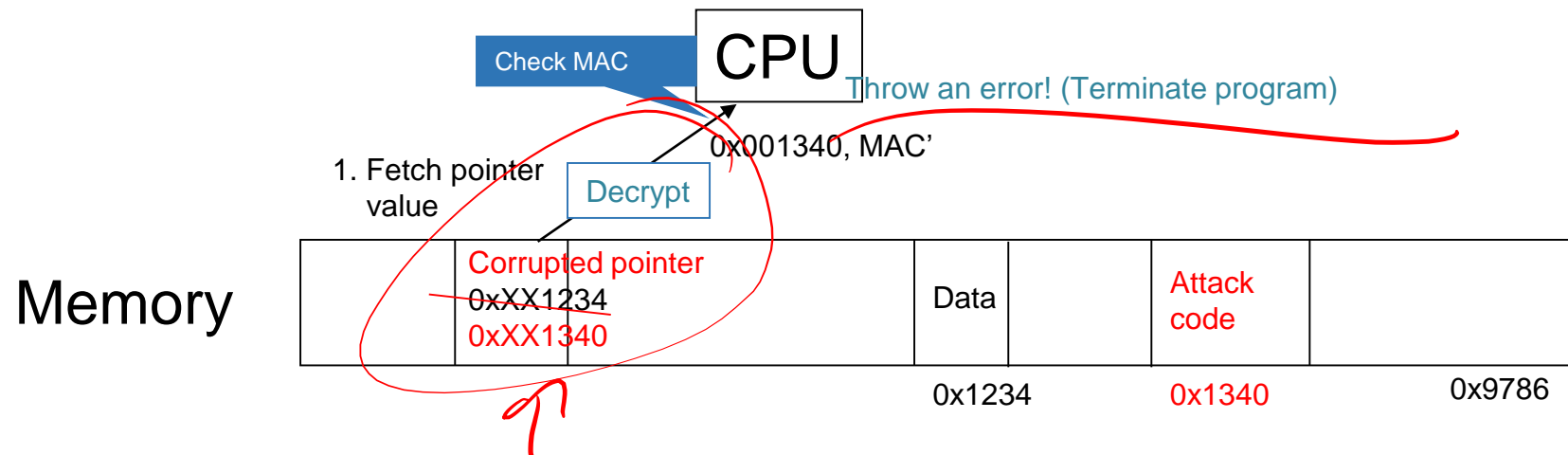
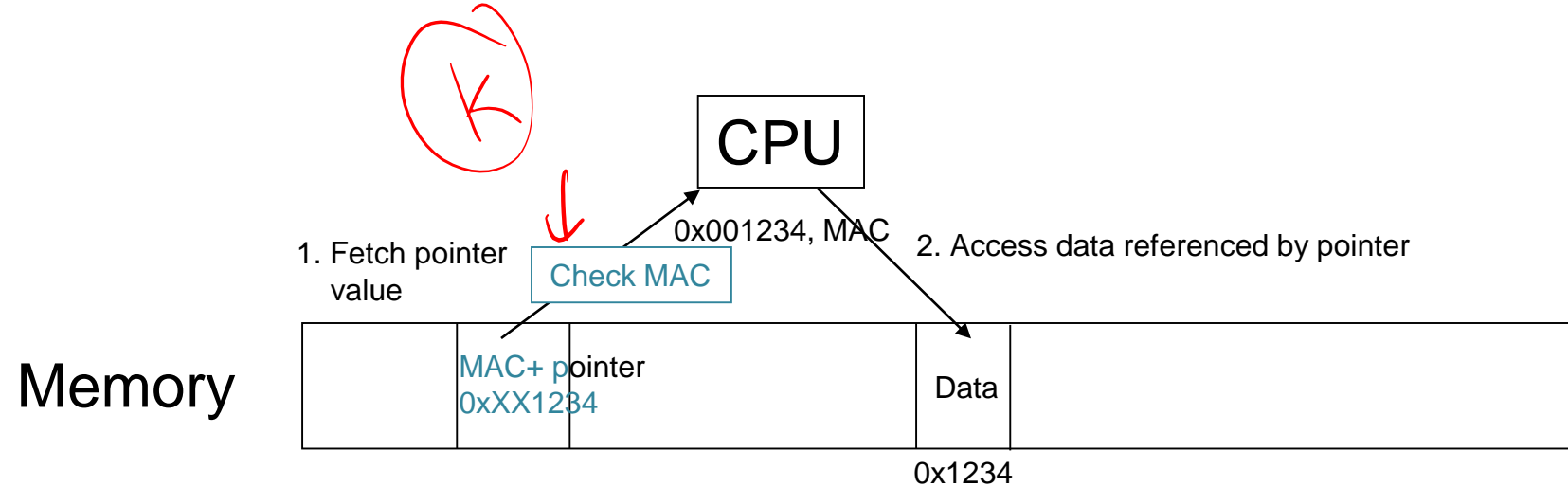
Pointer integrity protections (e.g. PointGuard, PAC, etc.)

- Attack: overwrite a pointer (heap data, ret, function pointer, etc.)
- Idea: encrypt all pointers while in memory
 - Generate a random key when program is executed
 - Each pointer is encrypted/XOR'd/MAC'd with this key when in memory
 - Pointers cannot be overflowed while in registers
- Attacker cannot predict the target program's key
 - If XOR/encrypt: adversary cannot predict what a corrupted pointer will do (mostly)
 - If integrity (MAC) then the program can *detect* a modified pointer.

Normal Pointer Dereference



Modern PAC Dereference



CFI: Control flow integrity

coop

call eax

- Idea: enforce branches to terminate 'where expected'
 - ... which is where? ↑

- Well, at the start of functions! →

- We shouldn't ever call into the middle of something!
- Put a special instruction at the start of every function: endbr64

- What about jumps (je, jz...)? →

- ... What about ret? ↑

exec: endbr64

for
set

Defense: Shadow stacks


- Idea: protect the *backwards edge* (return addresses on the stack)!
- Store them on... a **different stack**!
 - A *hidden* stack
- On function call/return
 - **Store/retrieve the return address from shadow stack**
- Or store on both main stack and shadow stack, and compare for equality at function return
- 2020/2021 Hardware Support emerges (e.g., Intel Tiger Lake, AMD Ryzen PRO 5000)



Challenges With Shadow Stacks

- Where do we put the shadow stack?
 - Can the attacker figure out where it is? Can they access it?
- How fast is it to store/retrieve from the shadow stack?
- How *big* is the shadow stack?
- Is this compatible with all software?
- (Still need to consider data corruption attacks, even if attacker can't influence control flow.)

Defense: Better string functions!

- strcpy is bad
- strncpy is... also bad (no null terminator! Returns dest!)


Defense: Better string functions!

- strcpy is bad
- strncpy is... also bad (no null terminator! Returns dest!)
- BSD to the rescue: ~~strcpy~~
 - `size_t strlcpy(char *dest, const char *src, size_t n);`
 - Always NUL terminates
 - Returns `len(src)`

Ushering out strlcpy()

By **Jonathan Corbet**
August 25, 2022

With all of the complex problems that must be solved in the kernel, one might think that copying a string would draw little attention. Even with the hazards that C strings present, simply moving some bytes should not be all that hard. But string-copy functions have been a frequent subject of debate over the years, with different variants being in fashion at times. Now it seems that the BSD-derived [`strlcpy\(\)`](#) function may finally be on its way out of the kernel.

What does a modern program do?

Normal, reasonable gcc config, (no optimization)

0000122d <foo>:

```

122d: f3 0f 1e fb      endbr32
1231: 55              push    %ebp
1232: 89 e5          mov     %esp,%ebp
1234: 53            push    %ebx
1235: 81 ec 34 01 00 00 sub     $0x134,%esp
123b: e8 b9 00 00 00  call    12f9 <__x86.get_pc_thunk.ax>
1240: 05 88 2d 00 00  add     $0x2d88,%eax
1245: 8b 55 08        mov     0x8(%ebp),%edx
1248: 89 95 d4 fe ff ff mov     %edx,-0x12c(%ebp)
124e: 65 8b 0d 14 00 00 00 mov     %gs:0x14,%ecx
1255: 89 4d f4        mov     %ecx,-0xc(%ebp)
1258: 31 c9          xor     %ecx,%ecx
125a: 8b 95 d4 fe ff ff mov     -0x12c(%ebp),%edx
1260: 83 c2 04        add     $0x4,%edx
1263: 8b 12          mov     (%edx),%edx
1265: 83 ec 08        sub     $0x8,%esp
1268: 52            push    %edx
1269: 8d 95 dc fe ff ff lea     -0x124(%ebp),%edx
126f: 52            push    %edx
1270: 89 c3          mov     %eax,%ebx
1272: e8 49 fe ff ff  call    10c0 <strcpy@plt>
1277: 83 c4 10        add     $0x10,%esp
127a: 90            nop
127b: 8b 4d f4        mov     -0xc(%ebp),%ecx
127e: 65 33 0d 14 00 00 00 xor     %gs:0x14,%ecx
1285: 74 05          je      128c <foo+0x5f>
1287: e8 f4 00 00 00  call    1380 <__stack_chk_fail_local>
128c: 8b 5d fc        mov     -0x4(%ebp),%ebx
128f: c9            leave
1290: c3            ret

```

Our custom gcc config

080491ad <foo>:

```

80491ad: 55              push    %ebp
80491ae: 89 e5          mov     %esp,%ebp
80491b0: 81 ec 18 01 00 00 sub     $0x118,%esp
80491b6: 8b 45 08        mov     0x8(%ebp),%eax
80491b9: 83 c0 04        add     $0x4,%eax
80491bc: 8b 00          mov     (%eax),%eax
80491be: 50            push    %eax
80491bf: 8d 85 e8 fe ff ff lea     -0x118(%ebp),%eax
80491c5: 50            push    %eax
80491c6: e8 95 fe ff ff  call    8049060 <strcpy@plt>
80491cb: 83 c4 08        add     $0x8,%esp
80491ce: 90            nop
80491cf: c9            leave
80491d0: c3            ret

```

Wait...

Attu/umnak's gcc config

```
080491ad <foo>:
80491ad: 55          push    %ebp
80491ae: 89 e5      mov     %esp,%ebp
80491b0: 81 ec 28 01 00 00  sub    $0x128,%esp
80491b6: 8b 45 08    mov     0x8(%ebp),%eax
80491b9: 83 c0 04    add     $0x4,%eax
80491bc: 8b 00      mov     (%eax),%eax
80491be: 83 ec 08    sub     $0x8,%esp
80491c1: 50          push    %eax
80491c2: 8d 85 e0 fe ff ff  lea     -0x120(%ebp),%eax
80491c8: 50          push    %eax
80491c9: e8 92 fe ff ff  call    8049060 <strcpy@plt>
80491ce: 83 c4 10    add     $0x10,%esp
80491d1: 90          nop
80491d2: c9          leave
80491d3: c3          ret
```

Our custom gcc config

```
080491ad <foo>:
80491ad: 55          push    %ebp
80491ae: 89 e5      mov     %esp,%ebp
80491b0: 81 ec 18 01 00 00  sub    $0x118,%esp
80491b6: 8b 45 08    mov     0x8(%ebp),%eax
80491b9: 83 c0 04    add     $0x4,%eax
80491bc: 8b 00      mov     (%eax),%eax
80491be: 50          push    %eax
80491bf: 8d 85 e8 fe ff ff  lea     -0x118(%ebp),%eax
80491c5: 50          push    %eax
80491c6: e8 95 fe ff ff  call    8049060 <strcpy@plt>
80491cb: 83 c4 08    add     $0x8,%esp
80491ce: 90          nop
80491cf: c9          leave
80491d0: c3          ret
```

Other Big Classes of Defenses

- Use safe programming languages, e.g., Java, Rust
 - What about legacy C code?
- Static analysis of source code to find overflows
- Dynamic testing: “fuzzing”

Fuzz Testing

- Generate “random” inputs to program
 - Sometimes conforming to input structures (file formats, etc.)
- See if program crashes
 - If crashes, found a bug
 - Bug may be exploitable
- Surprisingly effective
- Now standard part of development lifecycle