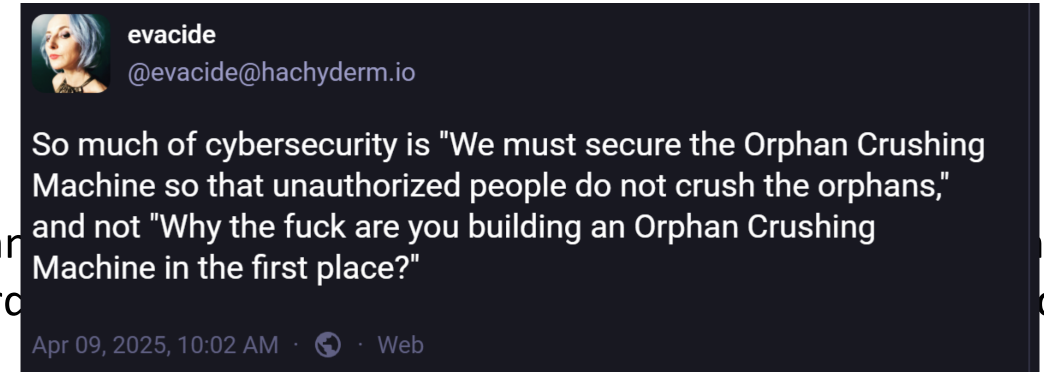
CSE 484/M584: Computer Security (and Privacy)

Spring 2025

David Kohlbrenner
dkohlbre@cs



UW Instruction Team: David Kohlbrenn anks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferd d many others for sample slides and materials

Admin

- Lab 1a due tonight!
- Late days
  - Can use up to 3 late days on one assignment. If you hand it in later, email David.
  - 5 total across homeworks and labs
- In-class assignments
  - As a reminder, 100% of these are required to do!
  - You may hand in _up to half_ by the late deadline (1 week later)
  - These do not interact with the lab/homework late days.
- HW1 out today
  - Writing and thinking about threat modeling

Threat modeling again again

- Microsoft announces a new feature: Recall!

- Recall:
  - AI assistant that runs locally on the machine, no cloud/etc.
  - Records most actions (e.g. screenshots every few seconds as you do things, records file accesses, edits, etc.)
  - Searchable so you can do things like ask: “What was that video I watched last wednesday about capybaras?” or “Which document had screenshots of the new UI design?”

Binary exploitation closeout

Summary of problems/techniques so far

- Classic overflow:
  - Unbounded (sploit 0/1) – Targeting saved return addresses
  - Limited overflow (sploit 2/3) – Targeting saved return addresses OR frame pointers

- Variable args/printf:

%x
%p
%s

  - Using % specifiers to read memory
    - Also to manipulate the internal argument pointer!

C

  - Using %n to *write* to a memory location
    - Remember it expects a pointer as argument!

Snprintf ?

Summary of using printf maliciously

- Printf takes a variable number of arguments
  - E.g., printf("Here's an int: %d", 10);
- Assumptions about input can lead to trouble
  - E.g., printf(buf) when buf="Hello world" versus when buf="Hello world %d"
  - Can be used to advance printf's internal argument pointer
  - Can read memory
    - E.g., printf("%x") will print in hex format whatever printf's internal argument pointer is pointing to at the time
  - Can write memory
    - E.g., printf("Hello%n"); will write "5" to the memory location specified by whatever printf's internal argument pointer is pointing to at the time

Heap buffer exploitation (exploit 5)

- Read “Once upon a free()” (linked in handout)
- Read through the tmalloc.c implementation
  - It is a complete malloc!
  - Manages things in ‘arena’

- Section will have exercises to walk you through the details

What is interesting about exploit 5?

- Advanced exploitation repurposes existing code to do something new, not what it was intended for.
- *“Weird machines”* is a recurring conceptual tool.
- Exploits 5 and 6 are a great introduction to this concept.

Other classic vulnerabilities

Another Class of Vulnerability:

0xFFFF, . . .

```c
char buf[80];
void vulnerable() {
    long long len = get_int_from_user();
    char *p = get_string_from_user();
    int32_t buflen = sizeof(buf);
    if (len > buflen) {
        error("length too large”);
         return;
    }
    memcpy(buf, p, len);
}
```

'\0'

Snippet 1

```c
size_t len = get_int_from_user();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

Snippet 2

```c
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

Implicit Cast

```
char buf[80];
void vulnerable() {
    long long len = get_int_from_user();
    char *p = get_string_from_user();
    int32_t buflen = sizeof(buf);
    if (len > buflen) {
        error("length too large”);
         return;
    }
    memcpy(buf, p, len);
}
```

Snippet 1

- If len is negative
- Then len > buflen may pass
- Any memcpy may copy huge amounts of input into buf.

```
void *memcpy(void *dst, const void * src, size_t n);
typedef unsigned int size_t;
```

Integer Overflow

- What if len is large (e.g., len = 0xFFFFFFFF)?
- Then len + 5 = 4 (on many platforms)
- Result: Allocate a 4-byte buffer, then read a lot of data into that buffer.

```c
size_t len = get_int_from_user();
char *buf;
buf = malloc(len+5);
read(fd, buf, len);
```

Snippet 2

```c
void *memcpy(void *dst, const void * src, size_t n);
```

```c
typedef unsigned int size_t;
```

Another Type of Vulnerability

- Consider this code, if run by the admin:

```
if (check_user_perms("file", WRITE_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, userbuffer, length(userbuffer));
```

- **Goal:** Write to file only with permission
- What can go wrong?

TOCTOU (Race Condition)

- TOCTOU = "Time of Check to Tile of Use"

```
if (check_user_perms("file", WRITE_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open("file", O_WRONLY);
write(fd, userbuffer, length(userbuffer));
```

- **Goal:** Write to file only with permission
- Attacker (in another program) can change meaning of "file" between access and open:

```
symlink("/etc/passwd", "file");
```

Something Different: Password Checker

- Functional requirements
  - PwdCheck(RealPwd, CandidatePwd) should:
    - Return TRUE if RealPwd matches CandidatePwd
    - Return FALSE otherwise
  - RealPwd and CandidatePwd are both 8 characters long

Password Checker

- Functional requirements
  - `PwdCheck(RealPwd, CandidatePwd)` should:
    - Return TRUE if RealPwd matches CandidatePwd
    - Return FALSE otherwise
  - RealPwd and CandidatePwd are both 8 characters long

i=0

i=7

- Implementation

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars always
  for(int i=0; i<8; i++){
      if (RealPwd[i] != CandidatePwd[i])
          return FALSE;
  }
  return TRUE;
```

Attacker Model

```
PwdCheck(RealPwd, CandidatePwd)  // both 8 chars always
    for(int i=0; i<8; i++){
        if (RealPwd[i] != CandidatePwd[i])
            return FALSE;
    }
    return TRUE;
```

AAAAAAAA

BAAA..

CAAA

CBAAA

CAAA

- Attacker can guess CandidatePwds through some standard interface
- Naive: Try all 256^8 = 18,446,744,073,709,551,616 possibilities
- Is it possible to derive password more quickly?

Try it

dkohlbre.com/cew

Hardening binaries

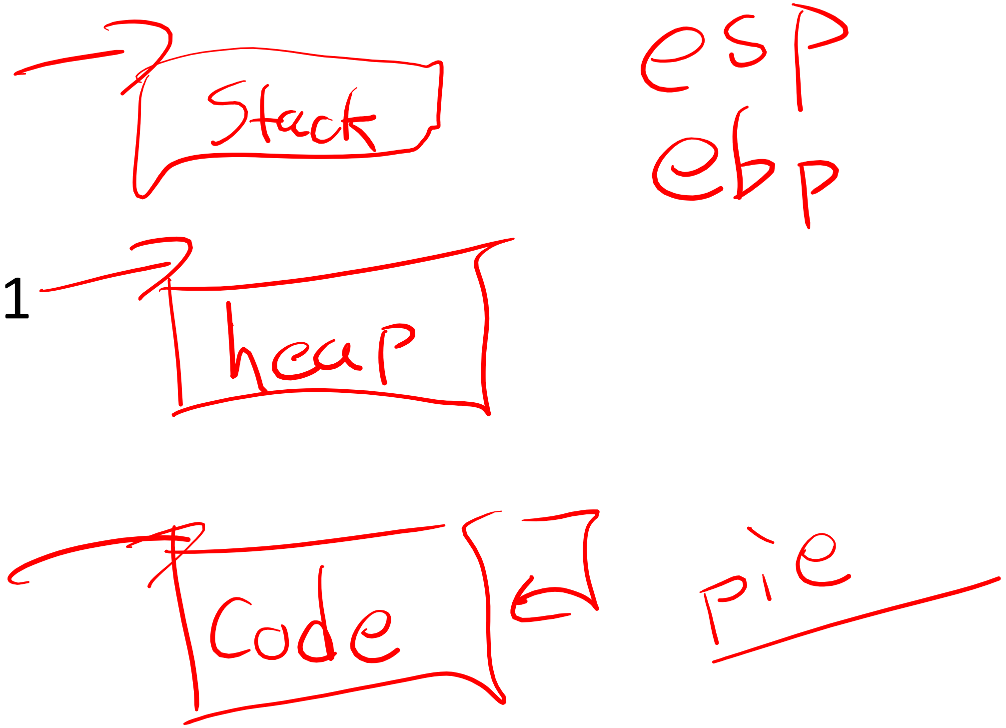
Buffer Overflow: Causes and Cures

- 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

Shellcode on stack

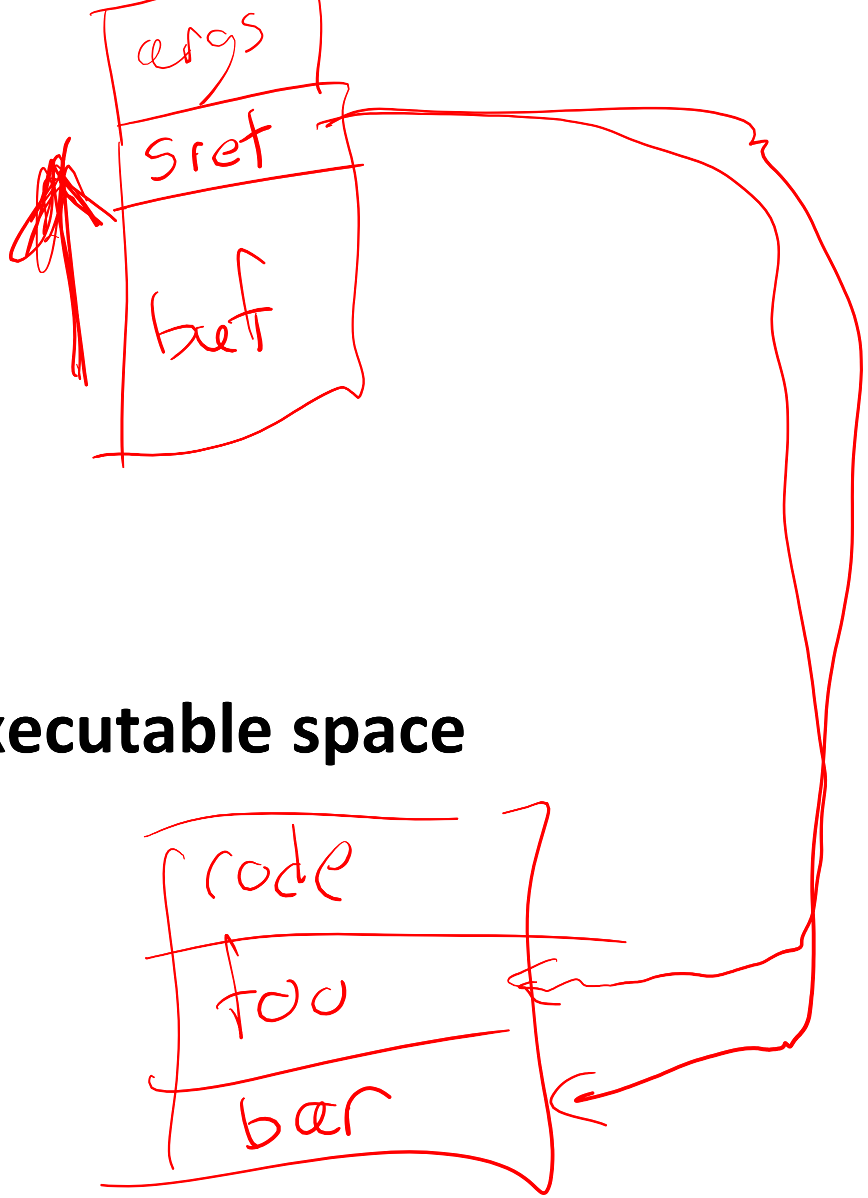
- Mark all writeable memory locations as non-executable
  - **This blocks many code injection exploits**
- Hardware support
  - AMD “NX” bit (no-execute), Intel “XD” bit (execute disable) (in post-2004 CPUs)
  - Makes memory page non-executable
- Widely deployed
  - Windows XP SP2+ (2004), Linux since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

RWX

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

strcpy

exec

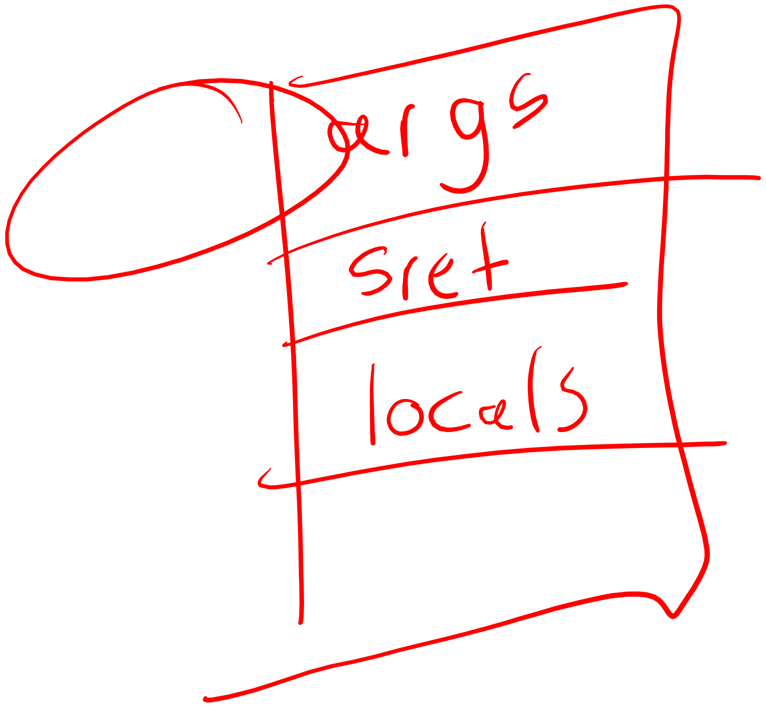
- Overwrite saved return address with address of any library routine

- Does not look like a huge threat?
  - ...

return-to-libc

exec

- 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++

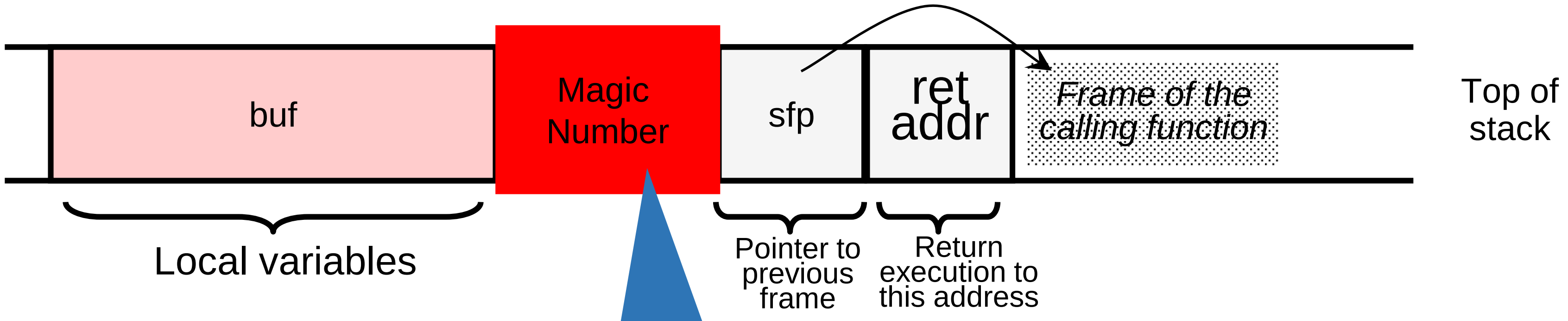
- Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read 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"

Defense: Run-Time Checking

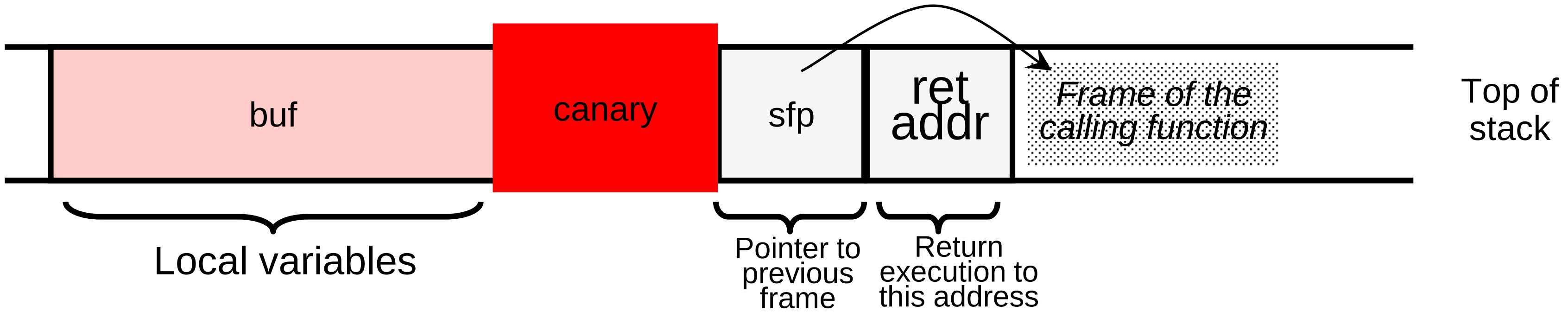
Gradescope: Why would this be useful? How could a program use this to protect against buffer overflows?



Choose randomly at the start of the program execution, keep constant during this program run.

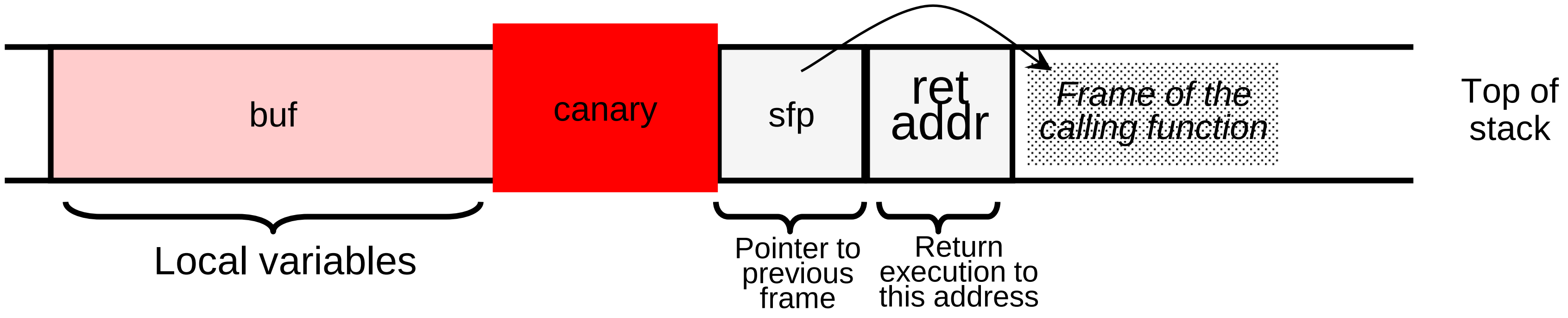
Defense: Run-Time Checking: StackGuard

- 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



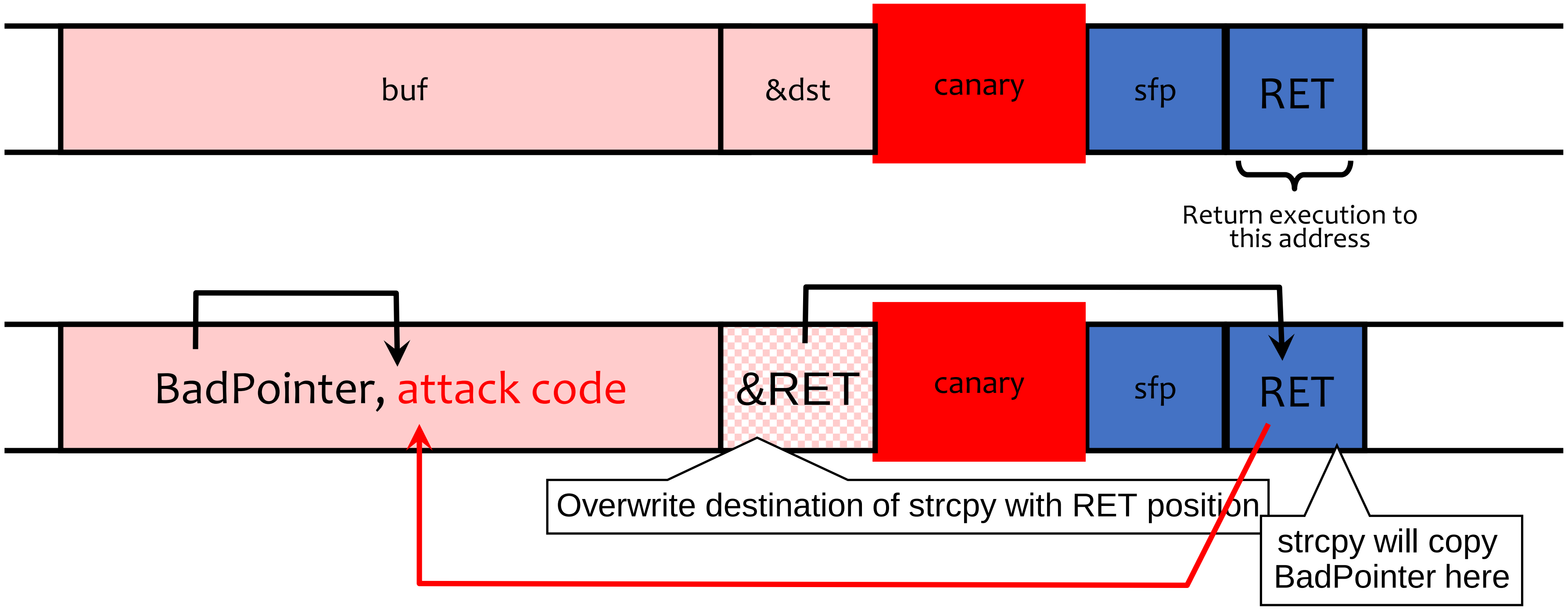
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Canary contains: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

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
- Suppose program contains copy(buf,attacker-input) and copy(dst,buf)
  - Example: dst is a local pointer variable
  - Attacker controls both buf and dst

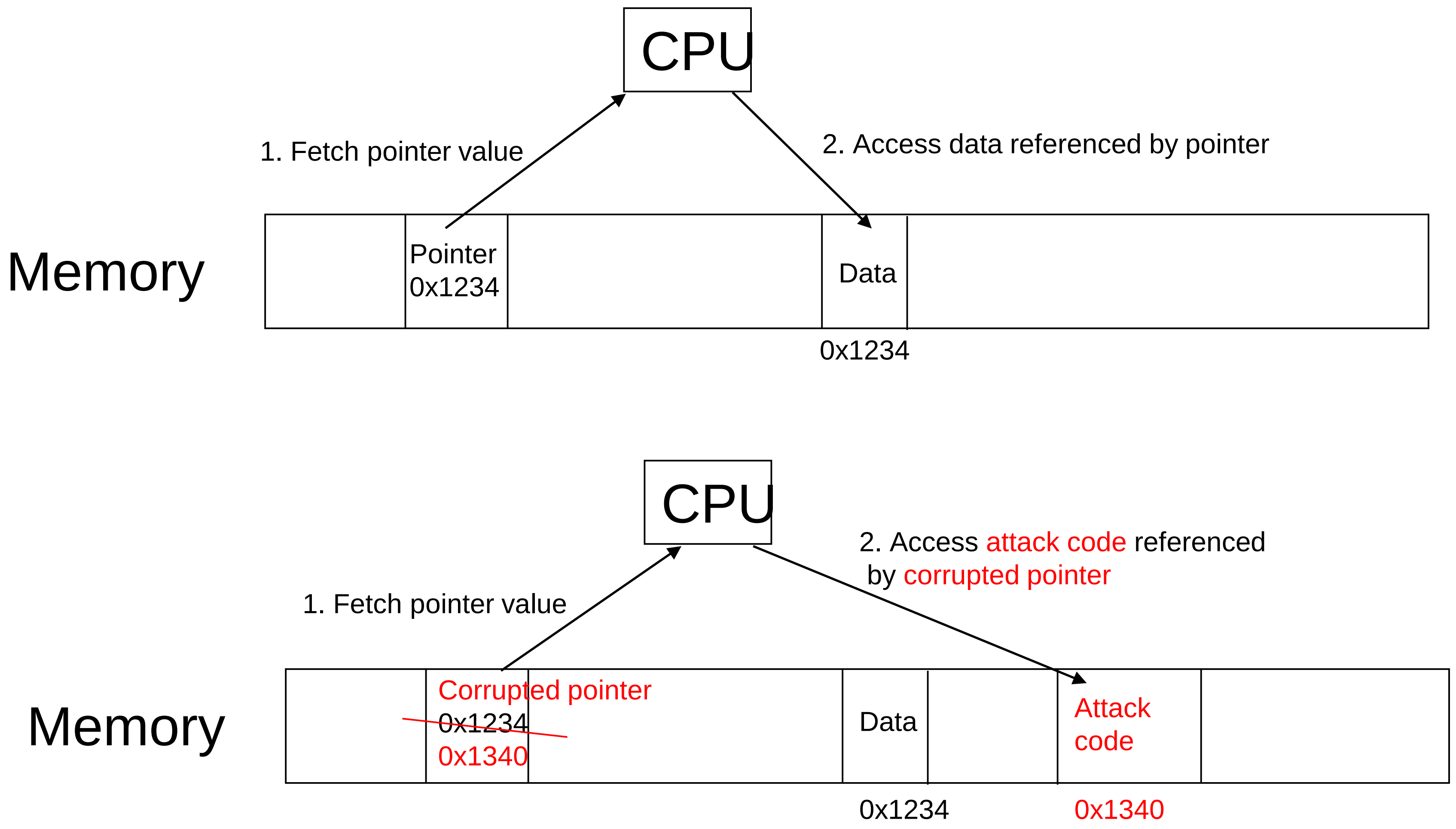


Pointer integrity protections (e.g. PointGuard, PAC, etc.)

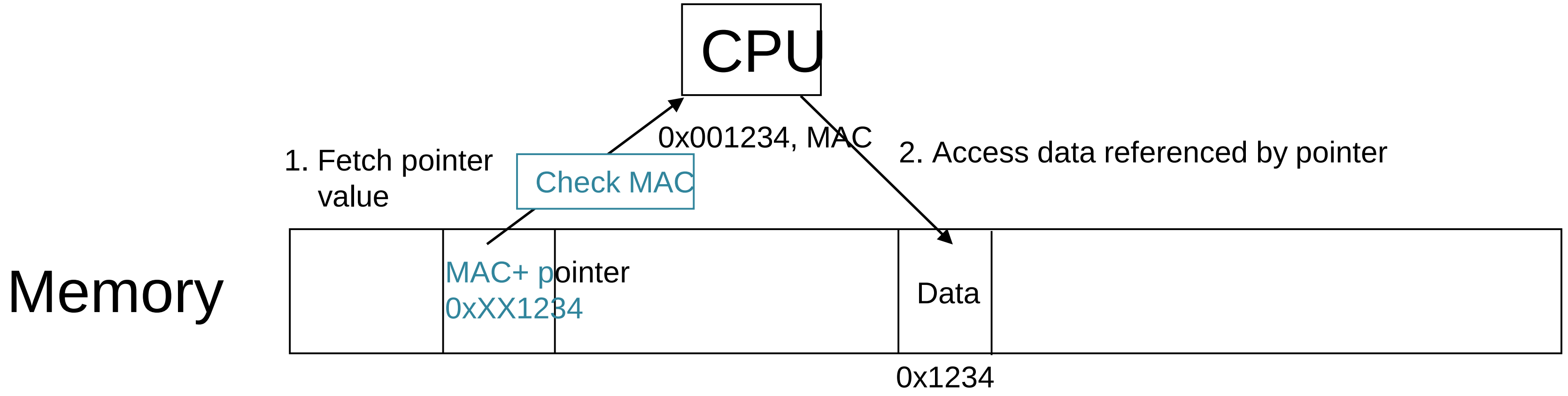
- Attack: overwrite a pointer (heap date, ret, function pointer, etc.)
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is encrypted/XOR'd/MAC'd with this key when in memory
    - Pointers cannot be overflowed while in registers
- Attacker 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.

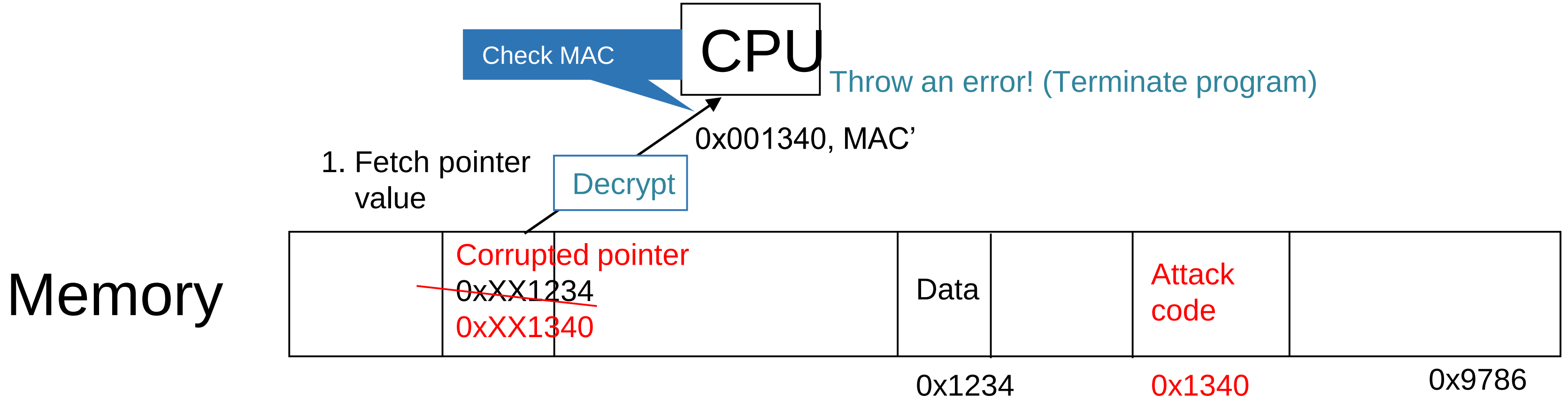
[Cowan]

Normal Pointer Dereference



Modern PAC Dereference





CFI: Control flow integrity

- 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`?

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: strlcpy
  - 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
```

T_T

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