CSE P564: Computer Security and Privacy

More Binary Exploitation (and Defenses)

Autumn 2024

David Kohlbrenner

dkohlbre@cs

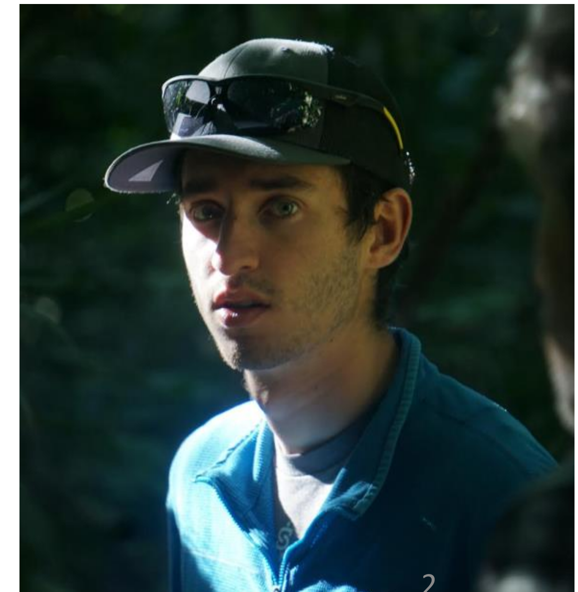
UW Instruction Team: David Kohlbrenner, Yoshi Kohno, Franziska Roesner. Thanks to Dan Boneh, Dieter Gollmann, Dan Halperin, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials

Office Hours

- Monday:
  - David
  - 5-6pm
  - Zoom (see Ed)
- Wednesday:
  - TAs
  - 5-6pm
  - In-person (Allen CSE1, 2nd floor)
- Friday:
  - TAs
  - 5-6+pm
  - Zoom (see Ed)

Kirandeep (Kiran) Kaur

Sela Navot



Paper discussion

Experimental Security Analyses of Automotive Attack Surfaces

But first!

- Did anyone note [14]?
  - Experimental Security Analysis of a Modern Automobile
  - Same authors :)

Pick one/more of the following to discuss

- What type of adversary does the paper focus on? (What are the threat model(s)?) Why?

- What were the biggest benefits or challenges the CAN bus system presents to the attacker?

- What seems like the major challenge of the paper?

- What is something you didn’t understand? (Maybe your conversation partner can help!)

Back to binary security

A note on assembly

- 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)

Stack Buffers

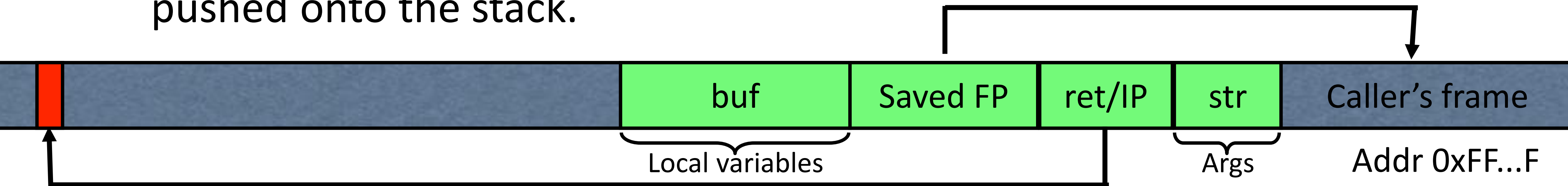
- Suppose Web server contains this function:

```
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.



What if Buffer is Overstuffed?

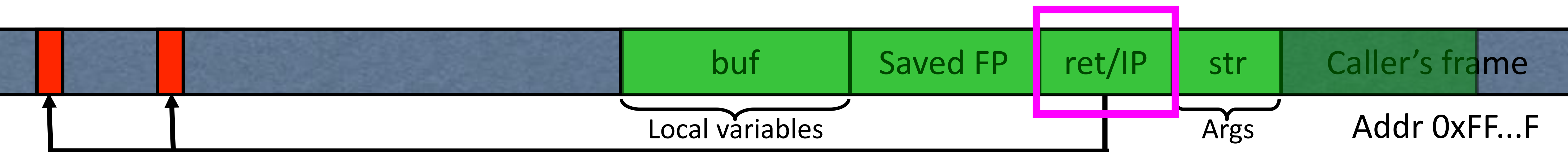
- Memory pointed to by str is copied onto stack…

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

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

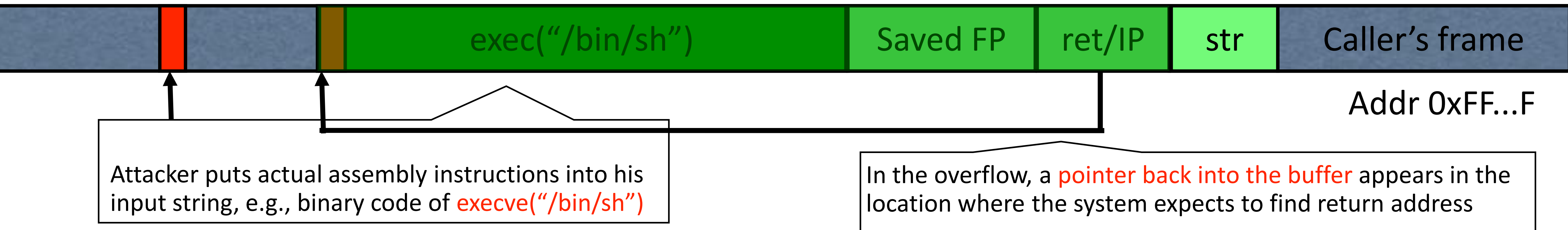
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

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



- 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…

- Overflow portion of the buffer must contain correct address of attack code in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
    - Otherwise application will (probably) crash with segfault
  - **Attacker must correctly guess in which stack position his/her buffer will be when the function is called**

Problem: No Bounds Checking

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

- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Lets do this live!

Break!

Does Bounds Checking Help?

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

- 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);
```

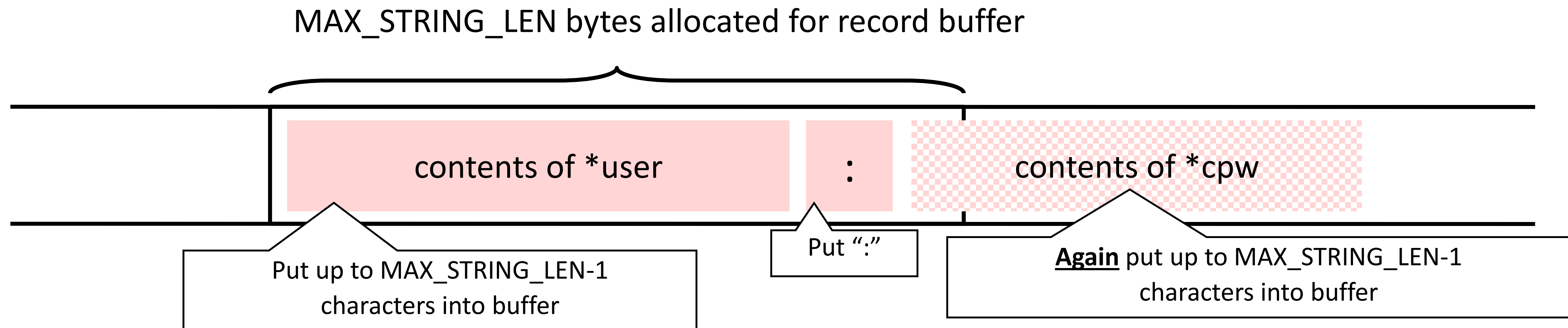
Breakout Activity

Gradescope!

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);
```



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]);
    }
```

Breakout Activity

Gradescope again!

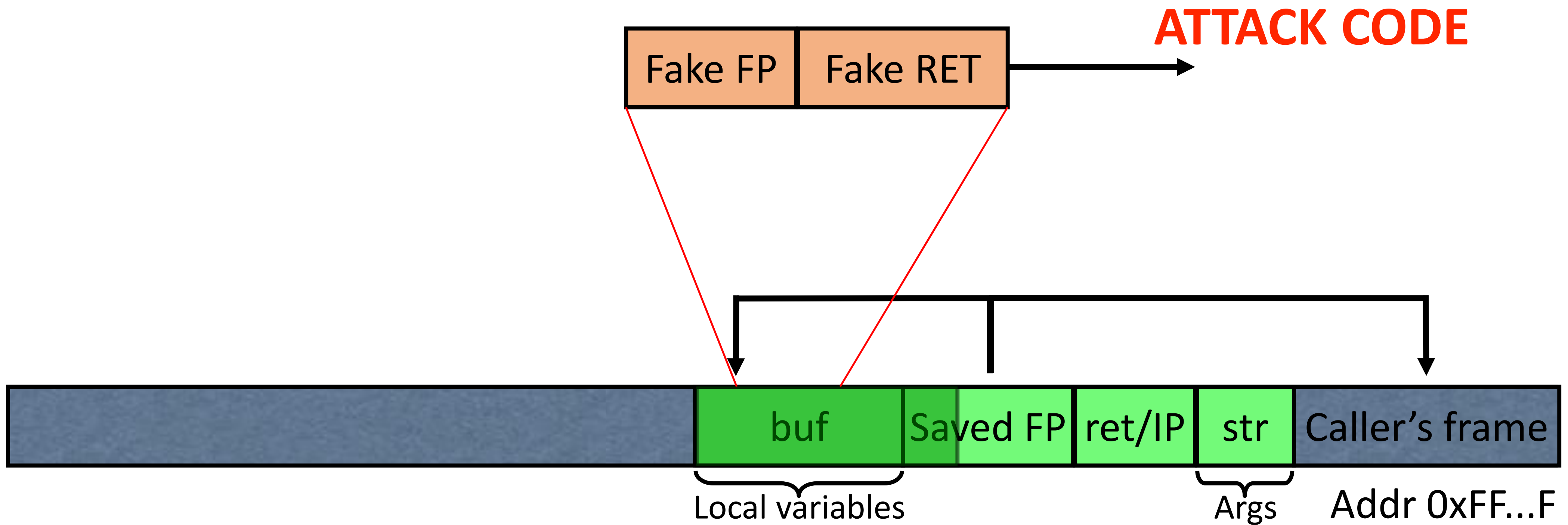
What About This? – Homebrew copy?

```
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!

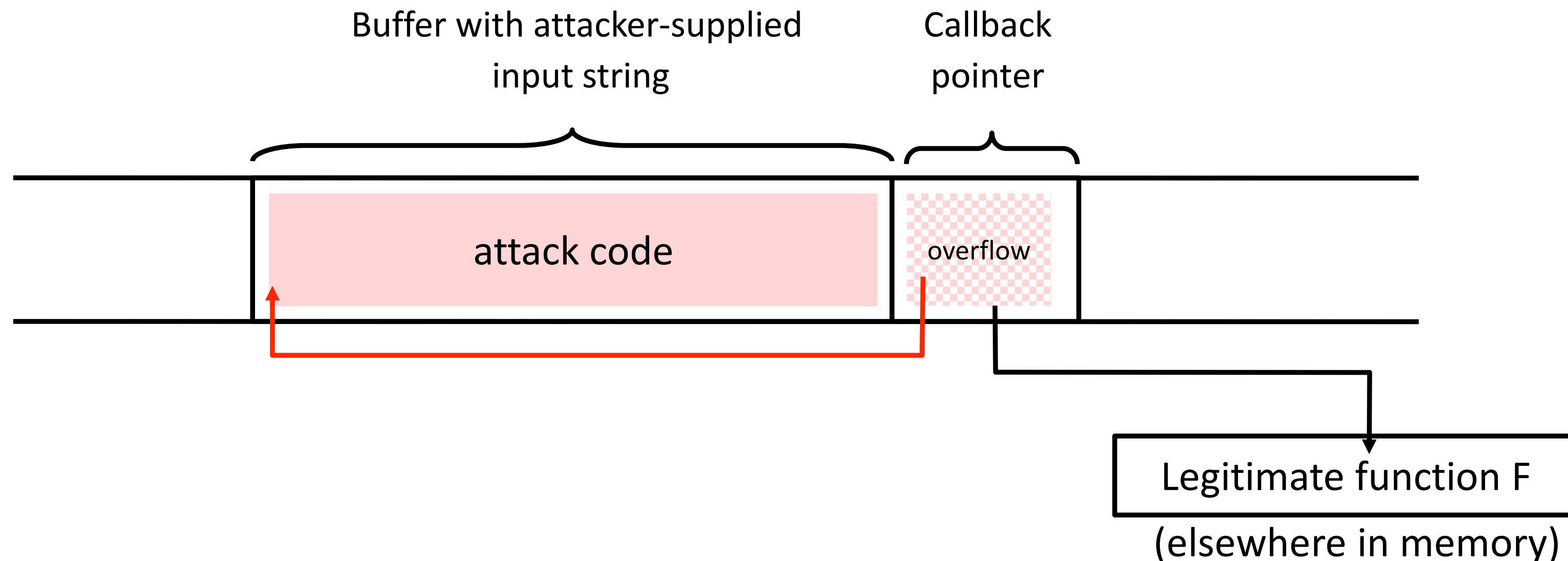
1-byte overflow: can’t change RET, but can change pointer to previous stack frame…

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)(…)



Variable arguments (and exploitation)

Variable Argument Functions in C (varargs)

- In C, can define a function with a variable number of arguments
  - E.g. `void printf(const char* format, …)`

- Examples of usage:

```
printf("hello, world");
printf("length of '%s' = %d\n", str, strlen(str));
printf("invalid fd %d\n", fd);
```

How does it know?

```
void printf(const char* format, …)
```

- printf uses ‘format’ to define what arguments are needed!
  - ‘%’ specifiers tell printf “expect another argument to this function
  - %i, %d, %x, etc. expect *integer* arguments
  - %p expects a *pointer* argument
  - %s expects a *pointer to a string* argument

Printf usage

- Reasonable usage

```
int foo = 1234;
printf("foo = %d in decimal, %X in hex", foo, foo);

"foo = 1234 in decimal, 4D2 in hex"
```

- Sloppy usage

```
char buf[14] = "Hello, world!";
printf(buf);
// should've used printf("%s", buf);
```

What happens if buf contains % format specifiers??

Implementation of Variable Args

- Special functions va_start, va_arg, va_end compute arguments at run-time

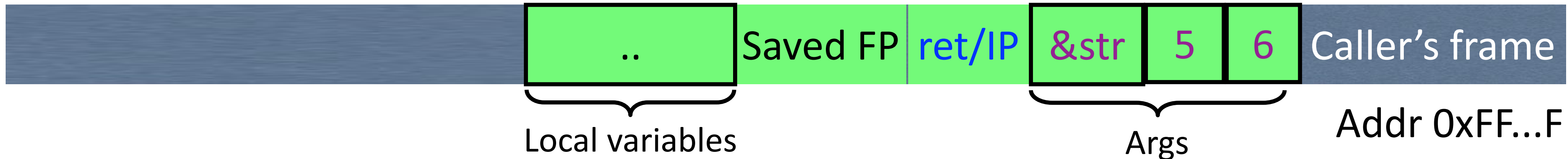
```
void printf(const char * format, ...)
{
  int i; char c; char * s; double d;
  va_list ap; /* declare an "argument pointer" to a variable arg list */
  va_start(ap, format); /*initialize arg pointer using last known arg */
  for (char *p = format; *p != '\0'; p++) {
    if (*p == '%') {
      switch (*++p) {
        case 'd':
          i = va_arg(ap, int); break;
        case 's':
          s = va_arg(ap, char*); break;
        case 'c':
          c = va_arg(ap, char); break;
        ... /* etc for each % specification */
      }
    }
  }
  ...
  va_end(ap); /* restore any special stack manipulations */
}
```

This is simplified code, e.g., handles %d but not %10d

Closer Look at the Stack

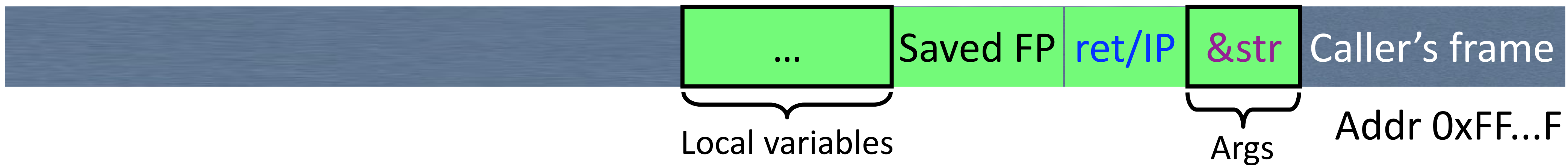
Internal argument pointer starts here

```
printf("Numbers: %d,%d", 5, 6);
```



Internal argument pointer starts here

```
printf("Numbers: %d,%d");
```



Printf usage

- Reaso

```
int
prin

“foo = 1234 in decimal, 4D2 in hex
```

If the buffer contains format symbols starting with %, the location pointed to by printf’s internal argument pointer will be interpreted as an argument of printf.

This can be exploited to move printf’s internal argument pointer!

- Sloppy usage

```
char buf[14] = “Hello, world!”;
printf(buf);
// should’ve used printf(“%s”, buf);
```

What happens if buf contains % format specifiers??

Reading memory with printf

- What happens if we pass an integer printing specifier?

```
printf(“Here is an int:  %x\n”, 25);
```

- What happens if we pass an integer printing specifier, but no arg?

```
printf(“Here is an int:  %x\n”);
```

- How about a string printing specifier?

```
printf(“Here is an string:  %s\n”);
```

Once again, lets do it live!

Writing Stack with Format Strings

- %n format symbol tells printf to write the number of characters that have been printed

```
int myVar = 0;
printf("Writing!%n",&myVar);
```

  - Argument of printf is interpreted as destination address
  - This writes 8 into myVar ("Writing!" has 8 characters)

- What if printf does not have an argument?

```
printf("Writing... somewhere!%n");
```

  - Stack location pointed to by printf's internal argument pointer will be **interpreted as address** into which the number of characters will be written.

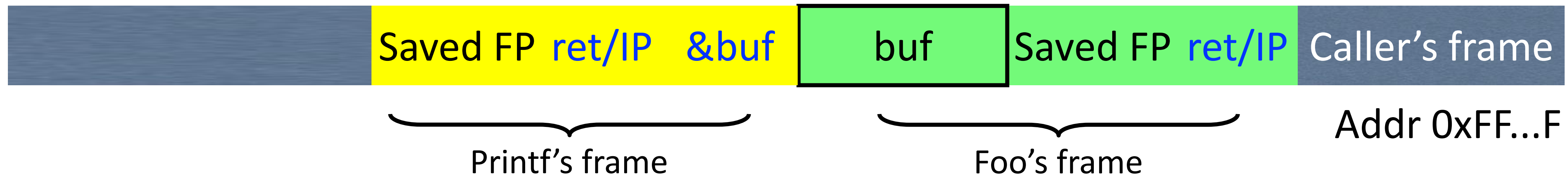
Summary of using printf maliciously

- 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

How Can We Attack This?

```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```

If format string contains % then printf will expect to find arguments here…

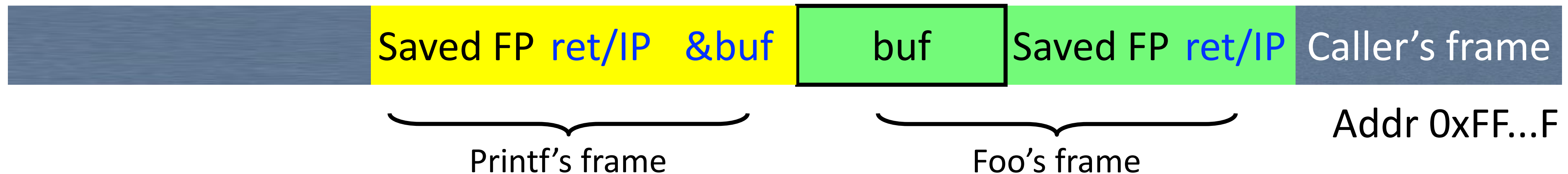


What should the string returned by `readUntrustedInput()` contain?

Different compilers / compiler options / architectures might vary

Discussion time!

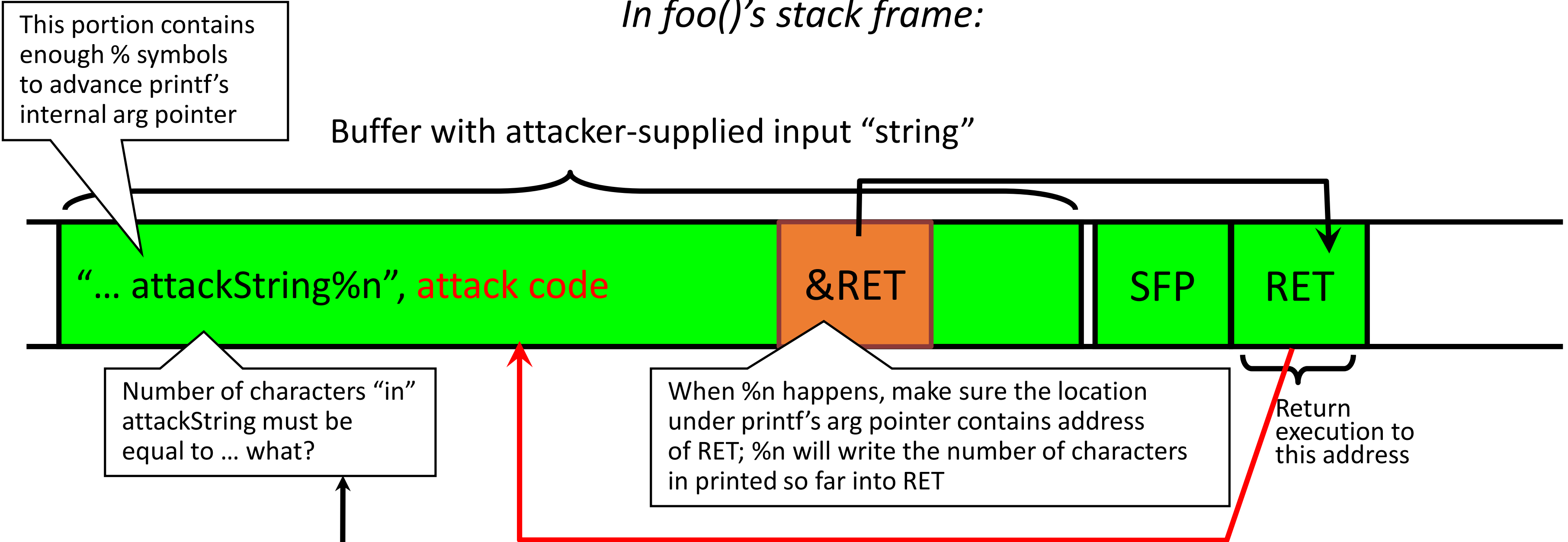
```
foo() {
    char buf[1024];
    strncpy(buf, readUntrustedInput(), sizeof(buf)-1);
    printf(buf); //vulnerable
}
```



What should the string returned by `readUntrustedInput()` contain?

Different compilers / compiler options / architectures might vary

Using %n to overwrite things



Why is "in" in quotes? C allows you to concisely specify the "width" to print, causing printf to pad by printing additional blank characters without reading anything else off the stack.

Example: printf("%5d%n", 10) will print three spaces followed by the integer: "   10"
That is, the %n will write 5, not 2.

Key idea: do this 4 times with the right numbers to overwrite the return address byte-by-byte. (4x %n to write into &RET, &RET+1, &RET+2, &RET+3)

The exploitation twilight zone

- During an exploitation attempt sometimes you have to ‘let it run’
  - Overflow a buffer
  - Change things
  - Let program run for ‘a bit’
  - Everything triggers!

- Printf exploit a perfect example

Break!

Heap buffer exploitation

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

Chunk header definition

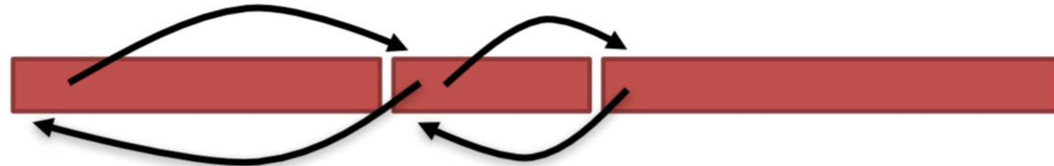
```
typedef union CHUNK_TAG
{
  struct
    {
      union CHUNK_TAG *l;       /* leftward chunk */
      union CHUNK_TAG *r;       /* rightward chunk + free bit (see below) */
    } s;
  ALIGN x;
} CHUNK;

/*
 * we store the freebit -- 1 if the chunk is free, 0 if it is busy --
 * in the low-order bit of the chunk's r pointer.
 */
```

Chunk Maintenance

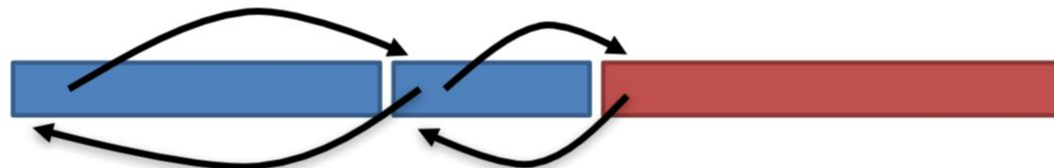
One big free chunk:

Split to malloc:

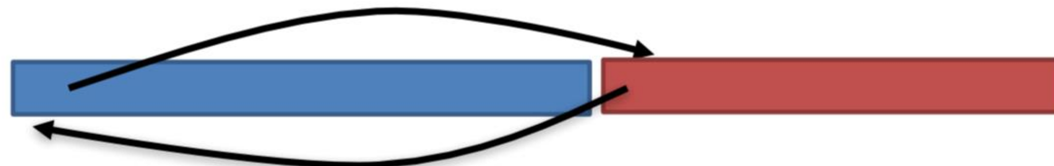
Split to malloc (twice):



Free (twice):



Consolidate free chunks:



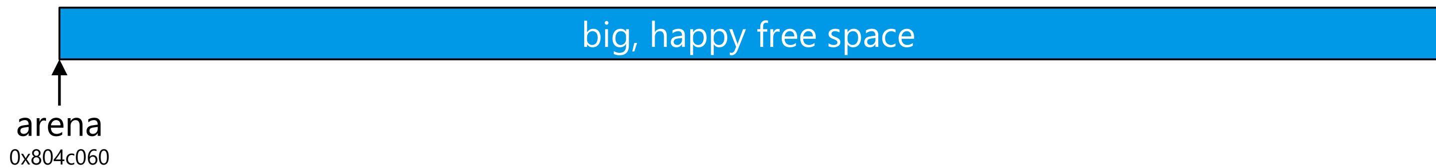
Refer to https://gitlab.cs.washington.edu/snippets/43 for a `tmalloc` implementation.

tmalloc.h usage example

```
1.  int main(){
2.      char* dyn;
3.      char* input =
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8
    \xa9";

4.      dyn = tmalloc(10);

5.      if(dyn == NULL){
6.          fprintf(stderr, "err\n");
7.          exit(EXIT_FAILURE);
8.      }

9.      memcpy(dyn, input, 10);

10.     tfree(dyn);

11.     return  0;
12. }
```

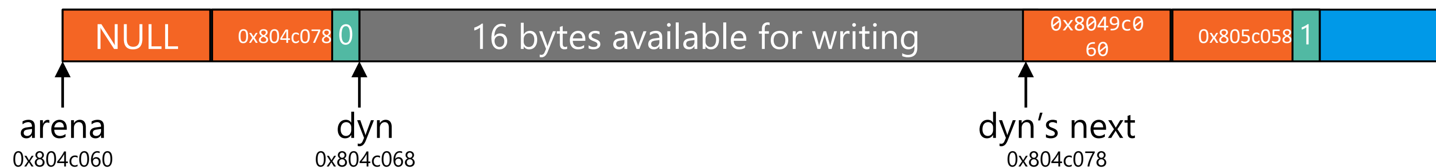
Before tmalloc call (line 4):

```
0x804c060 <arena>:     0x00000000   0x00000000   0x00000000   0x00000000
0x804c070 <arena+16>:  0x00000000   0x00000000   0x00000000   0x00000000
0x804c080 <arena+32>:  0x00000000   0x00000000   0x00000000   0x00000000
0x804c090 <arena+48>:  0x00000000   0x00000000   0x00000000   0x00000000
```



After tmalloc call: chunk pointers created

```
0x804c060 <arena>:     0x00000000   0x0804c078   0x00000000   0x00000000
0x804c070 <arena+16>:  0x00000000   0x00000000   0x0804c060   0x0805c059
0x804c080 <arena+32>:  0x00000000   0x00000000   0x00000000   0x00000000
0x804c090 <arena+48>:  0x00000000   0x00000000   0x00000000   0x00000000
```



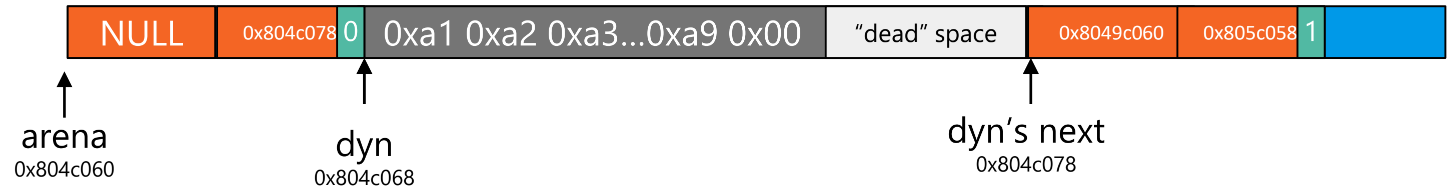
Printed with: x /16xw arena

tmalloc.h usage example

```
1.  int main(){
2.      char* dyn;
3.      char* input =
    "\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8
    \xa9";

4.      dyn = tmalloc(10);

5.      if(dyn == NULL){
6.          fprintf(stderr, "err\n");
7.          exit(EXIT_FAILURE);
8.      }

9.      memcpy(dyn, input, 10);

10.     tfree(dyn);

11.     return  0;
12. }
```

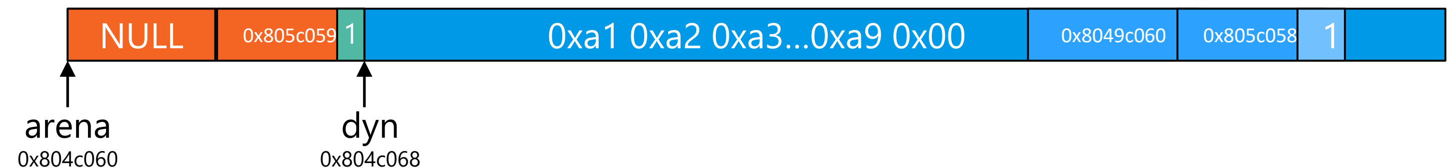
After the copy in line 9:

```
0x804c060 <arena>:      0x00000000  0x0804c078  0xa4a3a2a1  0xa9a8a7a5
0x804c070 <arena+16>:   0x000000a9  0x00000000  0x0804c060  0x0805c059
0x804c080 <arena+32>:   0x00000000  0x00000000  0x00000000  0x00000000
0x804c090 <arena+48>:   0x00000000  0x00000000  0x00000000  0x00000000
```



After the tfree, the chunk is coalesced (line 10)

```
0x804c060 <arena>:      0x00000000  0x0805c059  0xa4a3a2a1  0xa9a8a7a5
0x804c070 <arena+16>:   0x000000a9  0x00000000  0x0804c060  0x0805c059
0x804c080 <arena+32>:   0x00000000  0x00000000  0x00000000  0x00000000
0x804c090 <arena+48>:   0x00000000  0x00000000  0x00000000  0x00000000
```



Printed with: x /16xw arena

Double free?

- Target 5 makes a significant mistake!
  - It calls tfree(b) twice!
- The first call is fine, it does a “heap allocation free”
- The second call is…. What?
  - ????

Lets look at the code

Target 5

```c
int foo(char *arg)
{
  char *a;
  char *b;

  if ( (a = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (b = tmalloc(BUFLEN)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(a);
  tfree(b);

  if ( (a = tmalloc(BUFLEN * 2)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(a, arg, BUFLEN * 2);
  tfree(b);
  return 0;
}
```

Looking at tfree()

```
[…]
  p = TOCHUNK(vp);
  CLR_FREEBIT(p);
  q = p->s.l;
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
      CLR_FREEBIT(q);
      q->s.r      = p->s.r;
      p->s.r->s.l = q;
      SET_FREEBIT(q);
      p = q;
    }
[…]
```

Double free?

- Target 5 makes a significant mistake!
  - It calls tfree(b) twice!

- The first call is fine, it does a “heap allocation free”

- The second call is…. What?
  - *It isn’t logically a tfree()* it is just “some function”

Binary defenses

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. Stack “canaries”
  3. Encrypt pointers
  4. Address space layout randomization
  5. Code analysis
  6. Better interfaces
  7. …

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.

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)
- ASLR more effective on 64-bit architectures

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
  - Example: Microsoft's Data Execution Prevention (DEP)
  - **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 ret (IP) with address of any library routine

- Does not look like a huge threat?
  - …

- Gradescope time

return-to-libc

- Overwrite saved ret (IP) 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++

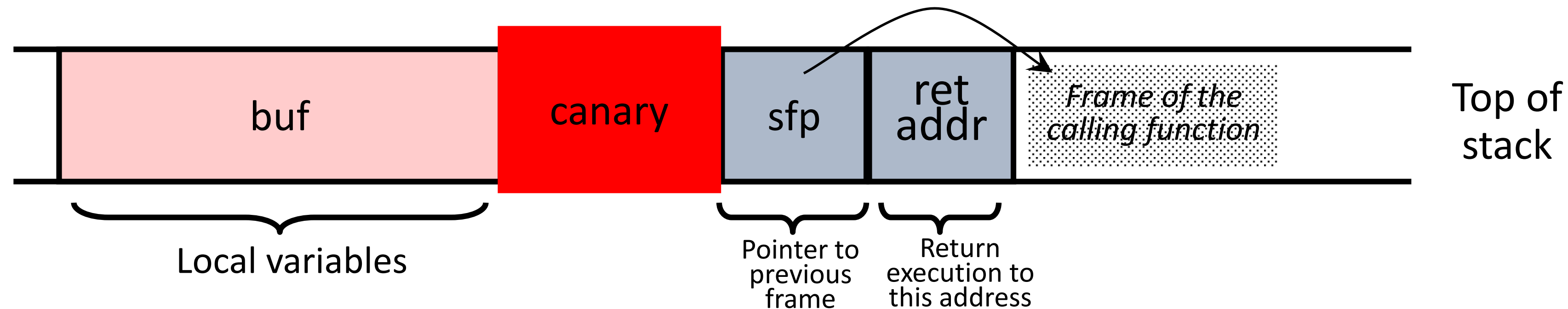
- Insight: Overwritten saved EIP need not point to the *beginning* of a library routine
- **Any** existing instruction in the code image is fine
  - Will execute the sequence starting from this instruction
- What if instruction sequence contains RET?
  - Execution will be transferred… to where?
  - Read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

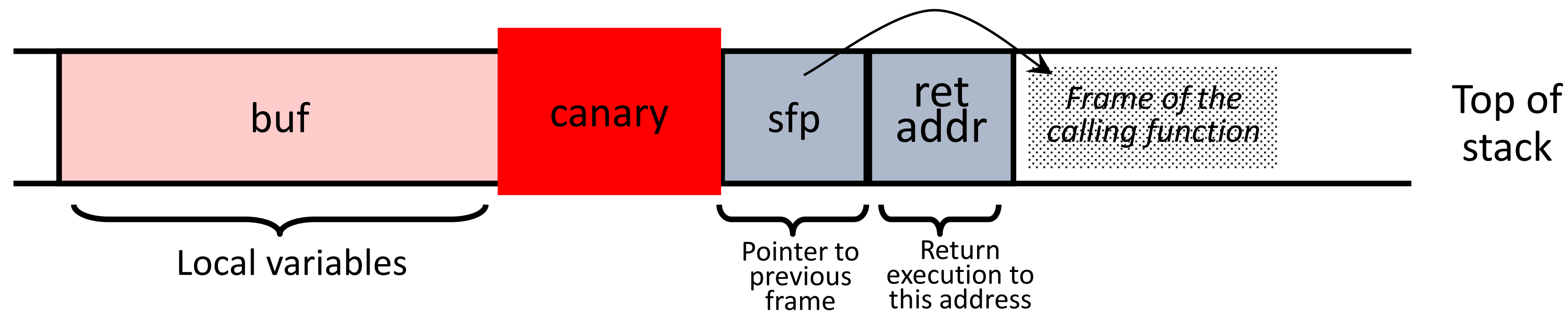
Defense: Run-Time Checking: StackGuard

- 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



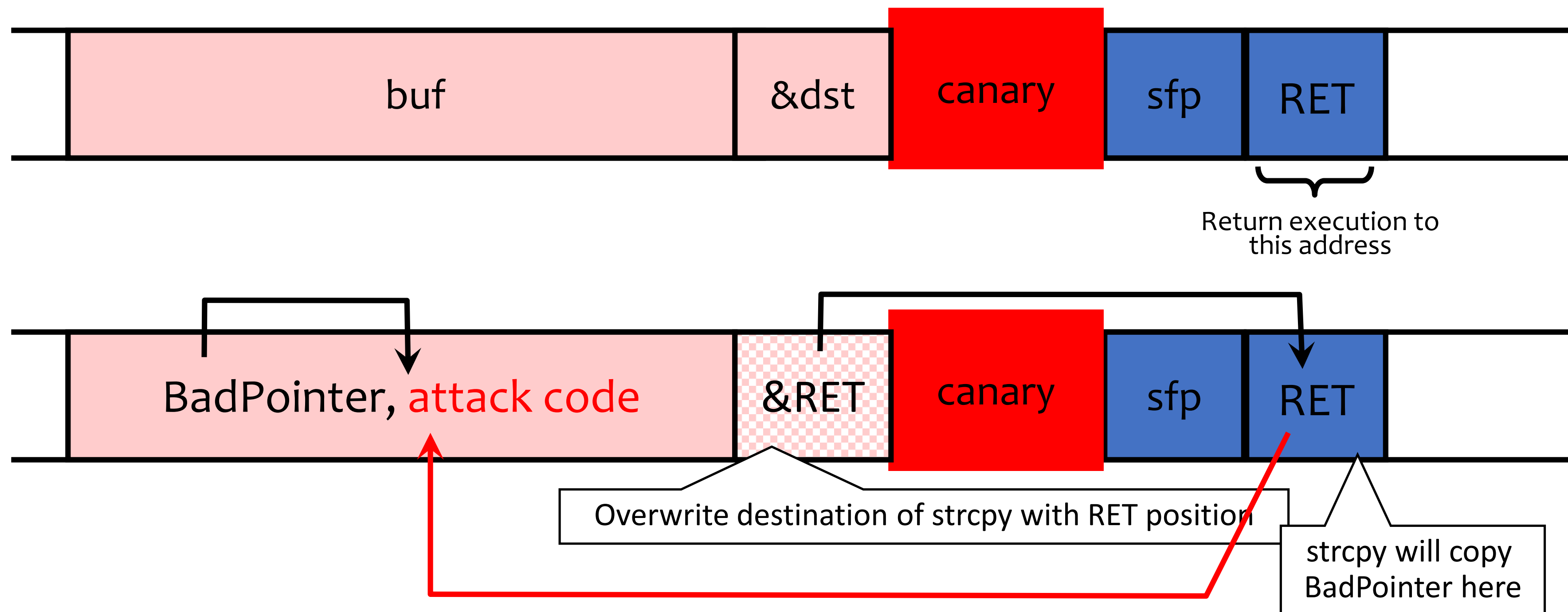
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Canary contains: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

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

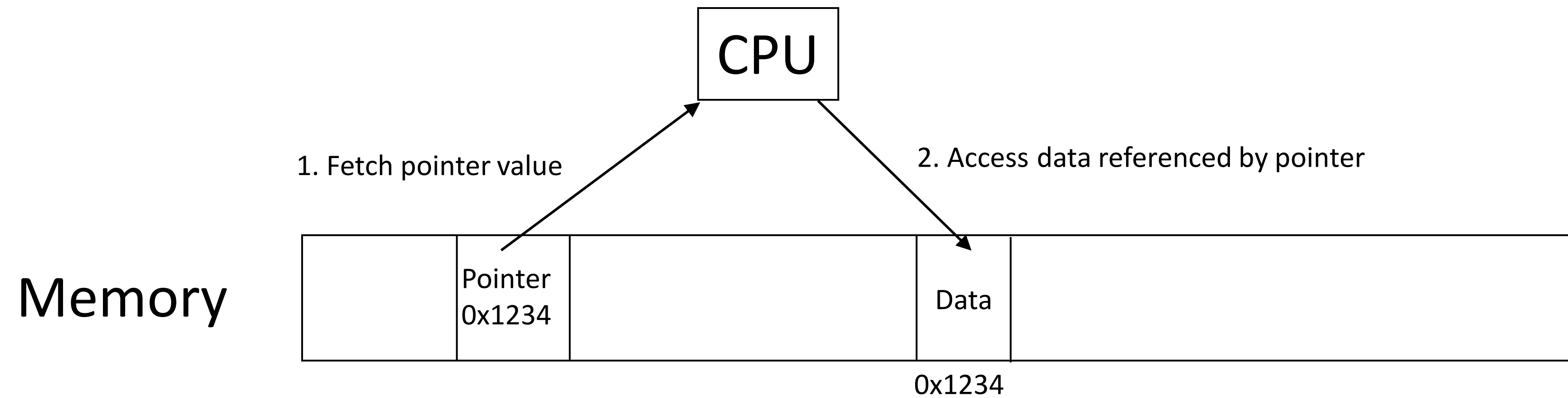


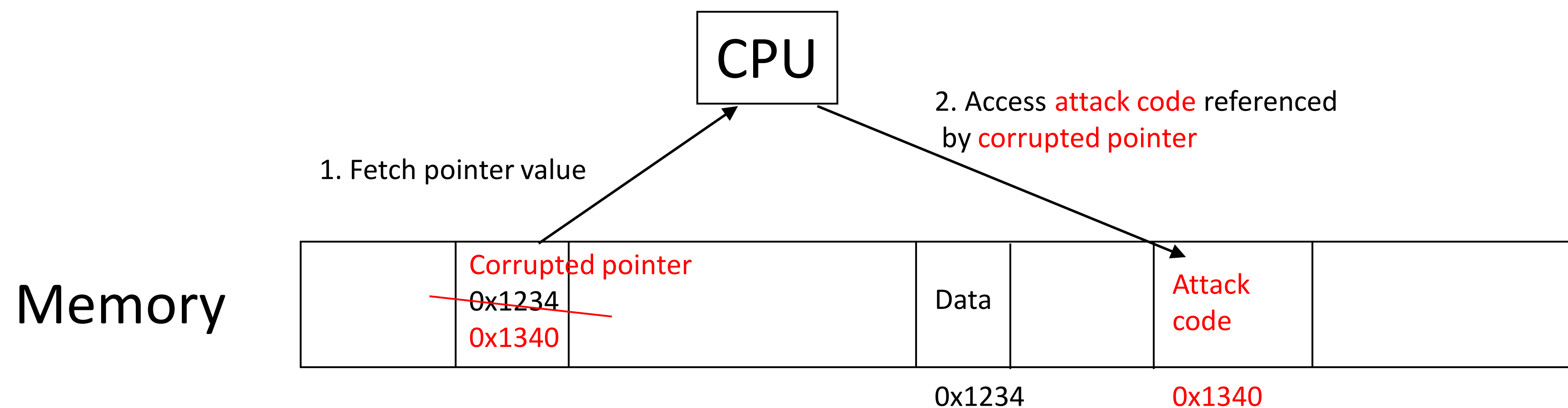
Pointer integrity protections (e.g. PointGuard, PAC, etc.)

- 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]

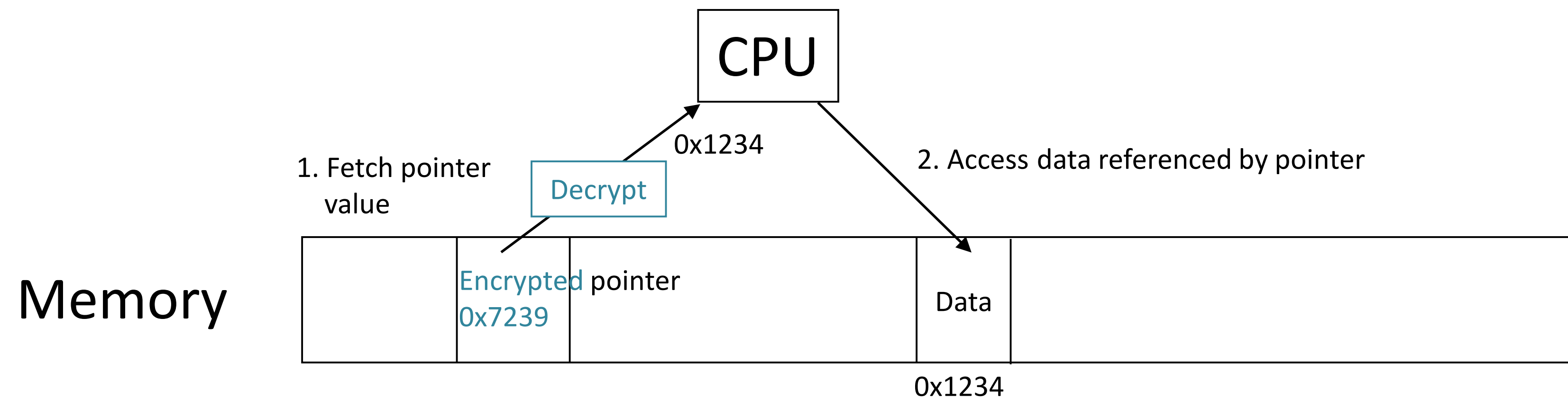
Normal Pointer Dereference

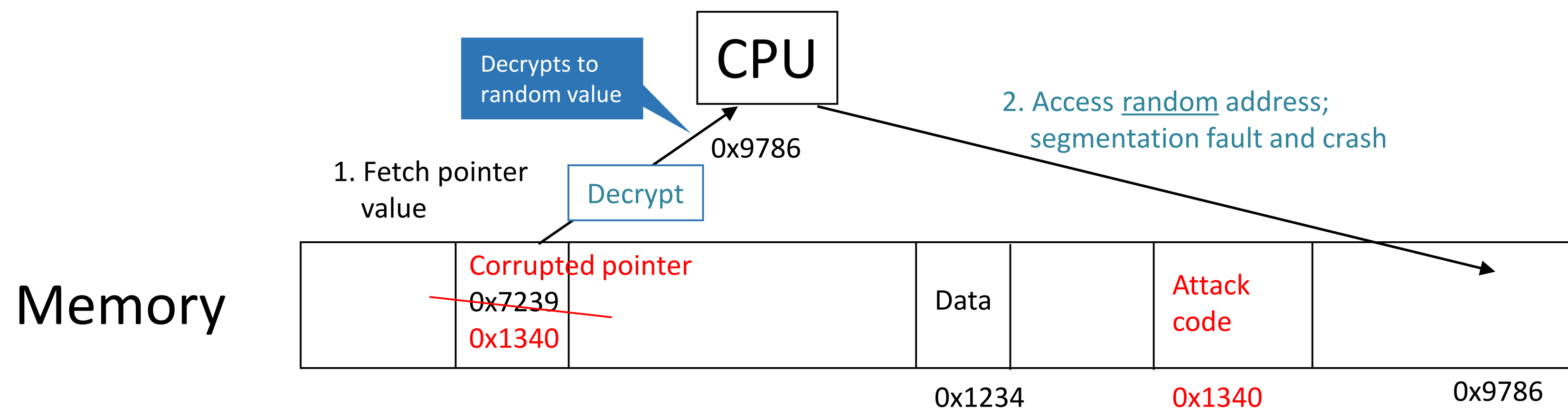




[Cowan]

PointGuard (Old, XOR style) Dereference



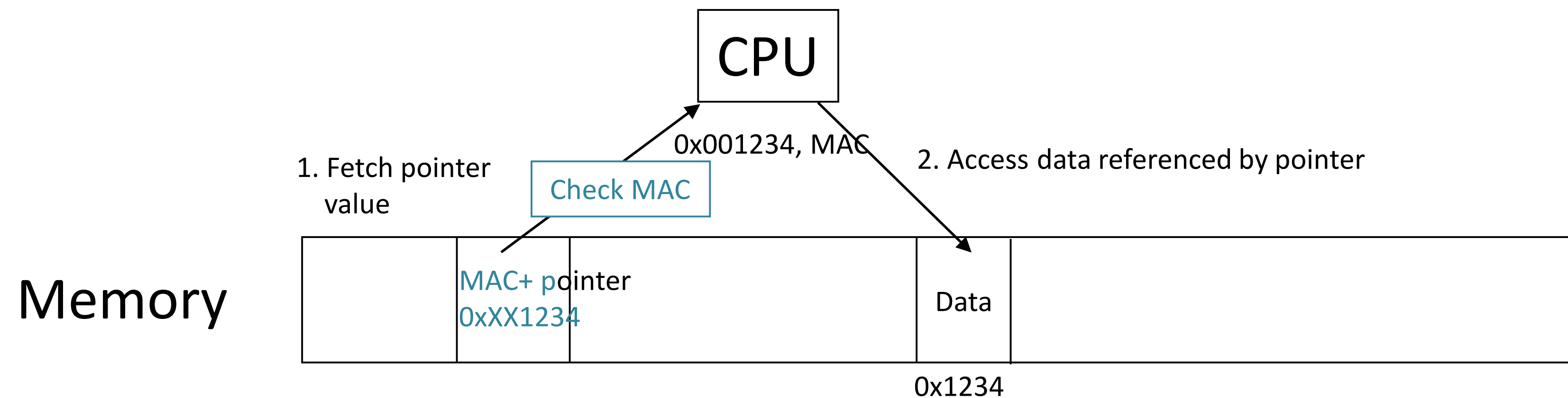


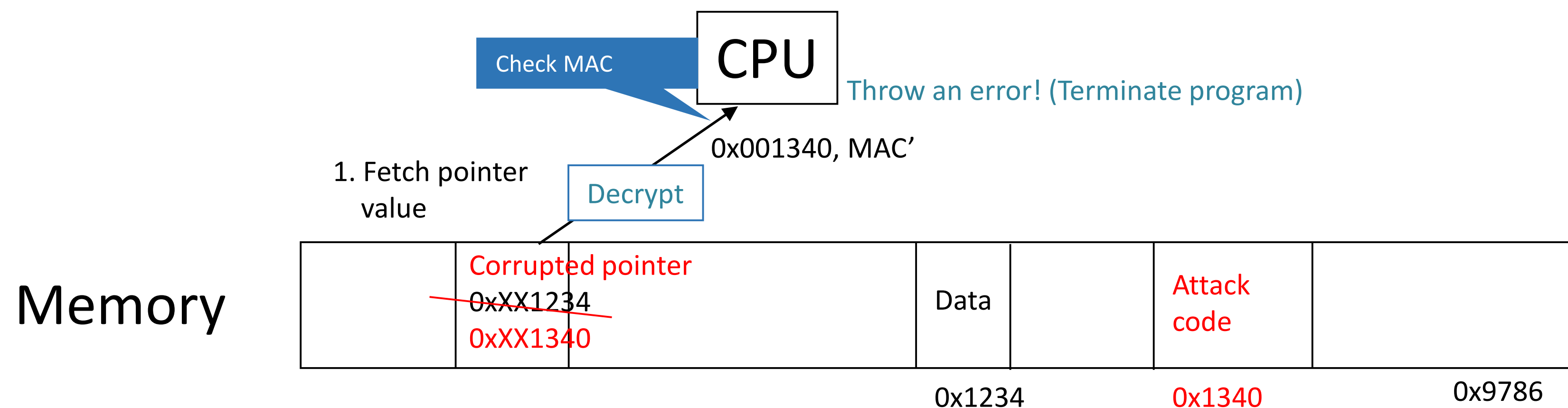
PointGuard Issues

- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt <u>only</u> pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

[Cowan]

Modern PAC Dereference





Defense: Shadow stacks

- Idea: don't store 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.)

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’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

More attack techniques

Other Common Software Security Issues…

Another Class of Vulnerability: (Gradescope)

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

Snippet 1

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

Snippet 2

```
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_attacker();
     char *p = get_string_from_attacker();
     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.

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

Snippet 2

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

```
typedef unsigned int size_t;
```

Another Type of Vulnerability

- Consider this code:

```
if (access(“file”, W_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open(“file”, O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

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

TOCTOU (Race Condition)

- TOCTOU = “Time of Check to Tile of Use”

```
if (access(“file”, W_OK) != 0) {
    exit(1); // user not allowed to write to file
}

fd = open(“file”, O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

- **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
- Implementation (like TENEX system)

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

- Clearly meets functional description

Attacker Model

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

- 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