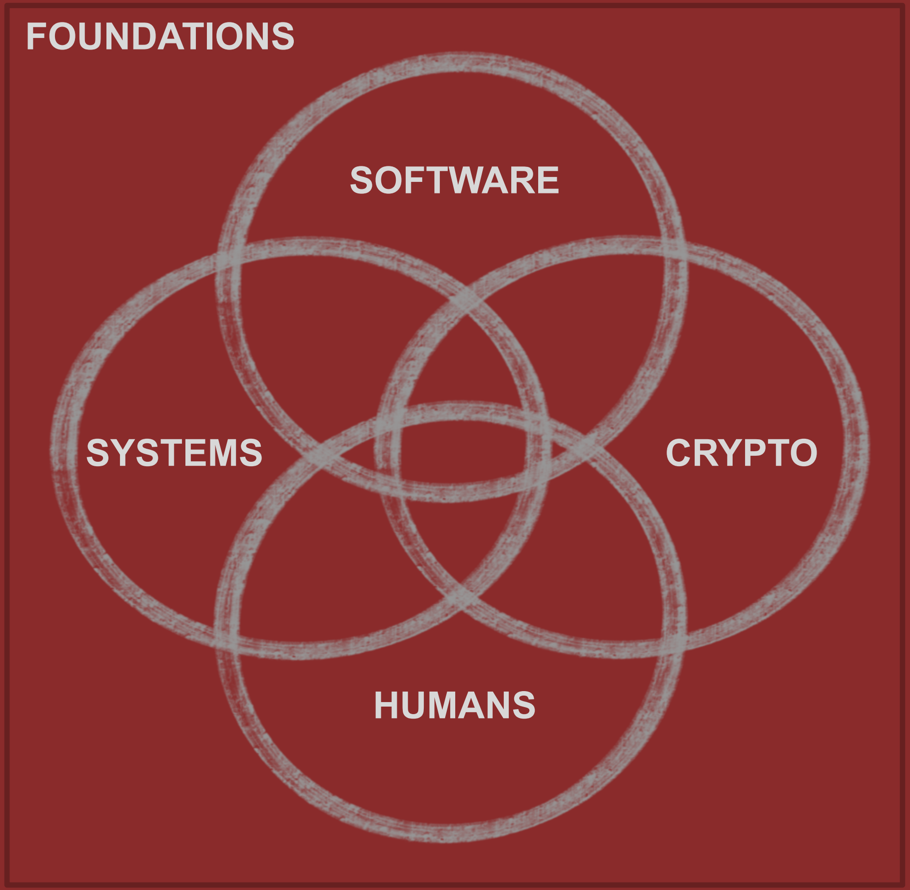
Διάλεξη #4 - Format String Attacks



Huge thank you to David Brumley from Carnegie Mellon University for the guidance and content input while developing this class

Την Προηγούμενη Φορά

1. Control Flow Hijack Attacks
2. Basics of buffer overflow attacks continued (shellcode + nopsled)
3. x86 Fundamentals continued

Ανακοινώσεις / Διευκρινίσεις

- Την Δευτέρα κλείνει το Μπόνους #0

Σήμερα

- Variadic Functions
- Format String Attacks

Variadic Functions

Variadic Συναρτήσεις

Συναρτήσεις που έχουν μεταβαλλόμενο αριθμό ορισμάτων (π.χ., printf) λέγονται variadic. Δηλώνουμε τον μεταβλητό αριθμό ορισμάτων χρησιμοποιώντας την έλλειψη (ellipsis): …

Παραδείγματα:

```
int printf(const char * format, ...);

int scanf(const char * format, ...);
```

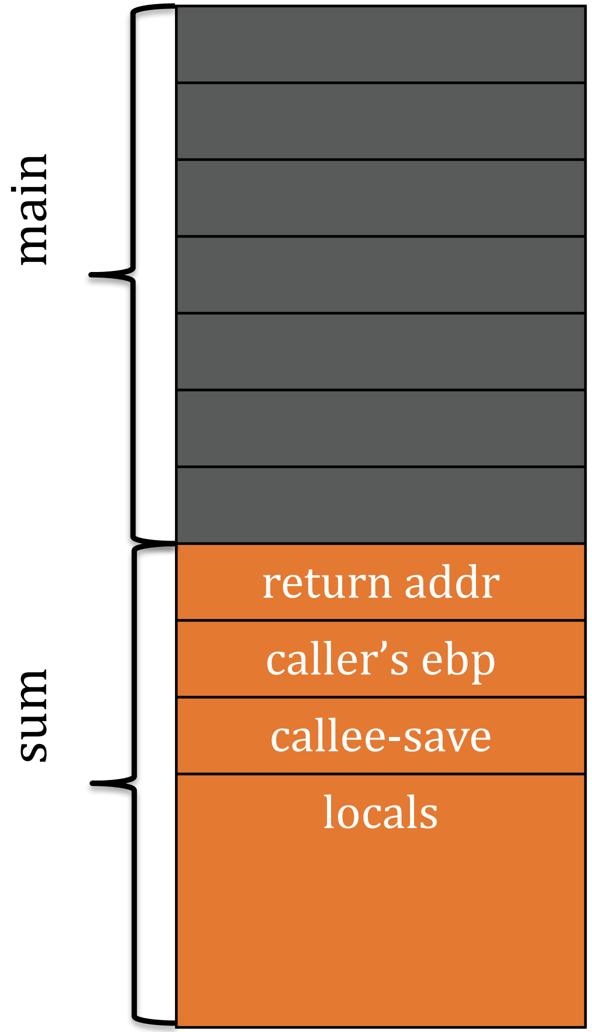
Παράδειγμα με Variadic Function

```
#include <stdarg.h>

int sum(int count, ...) {

  int result = 0;

  va_list args;

  va_start(args, count);

  for(int i = 0; i < count; i++)

     result += va_arg(args, int);

  va_end(args);

  return result;

}

int main() {

  return sum(6, 1, 2, 3, 4, 5, 6);

}
```

```
$ ./variadic
$ echo $?
21
```

Χρησιμοποιώντας τις "μαγικές" συναρτήσεις va_list, va_start, va_arg, va_end μπορούμε να διατρέξουμε όλα τα ορίσματα (δεν ξέρουμε τι κάνουν; χρησιμοποιούμε man!)

Stack Diagram



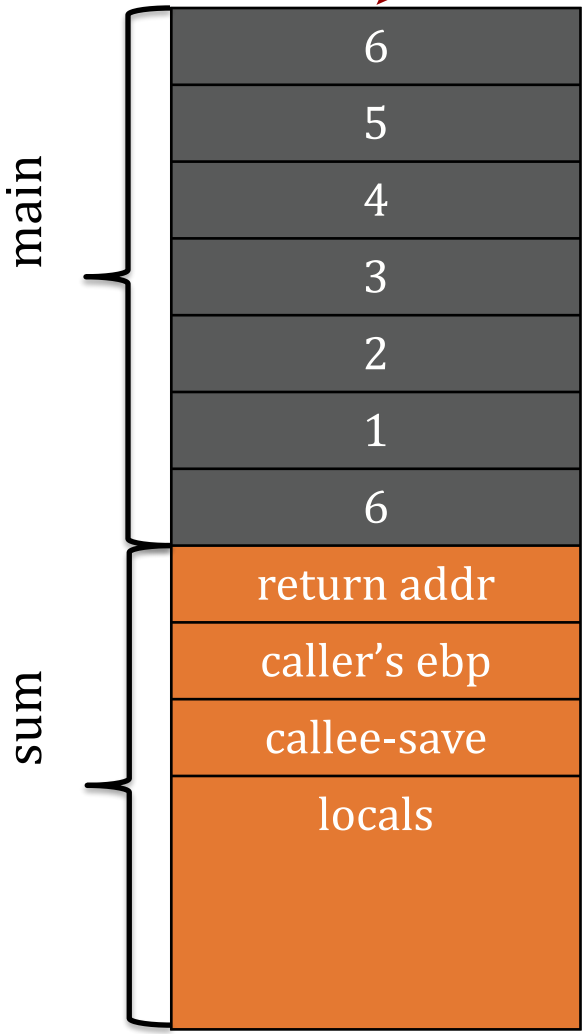
Μόλις καλέσαμε την

`sum(6, 1, 2, 3, 4, 5, 6)`

Πως είναι το stack;

Stack Diagram

Each cell 4 bytes



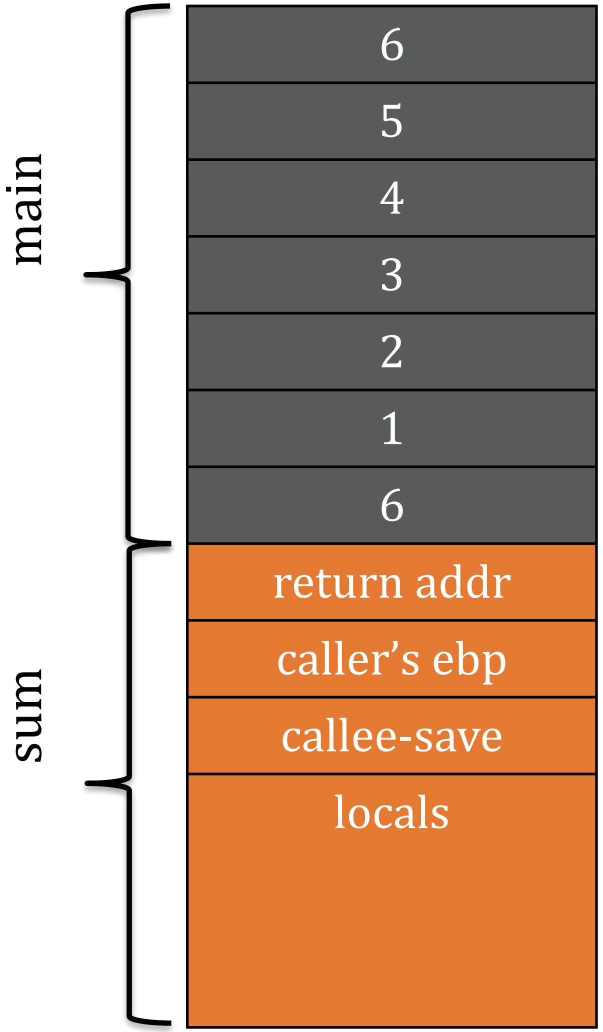
7th argument

1st argument

Why do you think va_list is initialized with "count"? Careful, this is a trick question!

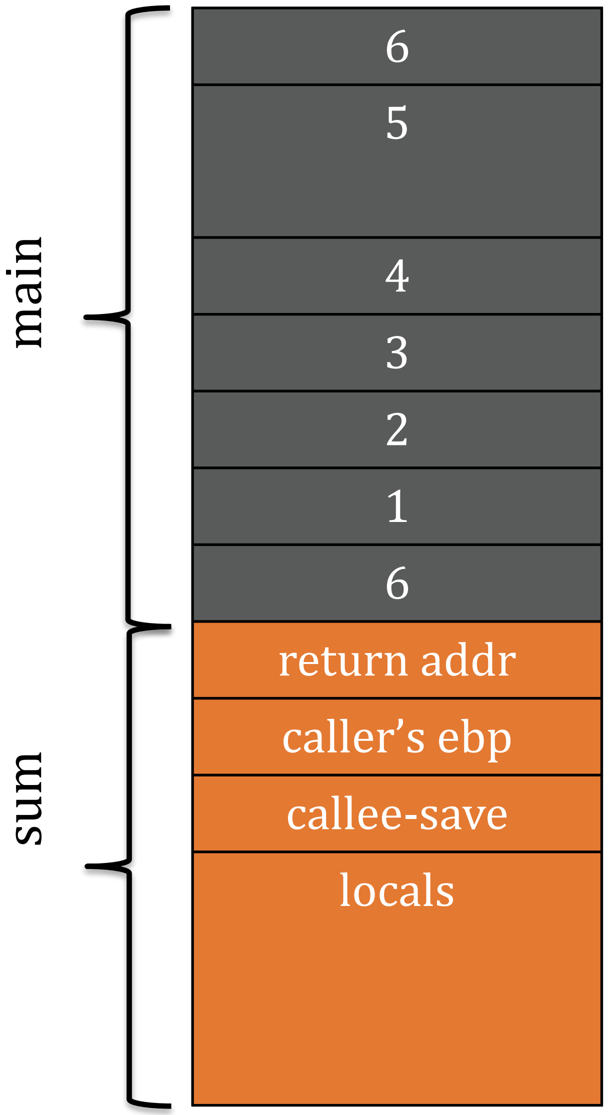
```
va_list args;

va_start(args, count);

for(int i = 0; i < count; i++)
```

Stack Diagram



What would happen if the argument corresponding to **5** was of type int64_t?

Stack Diagram



What would happen if the argument corresponding to **5** was of type int64_t?

What about the va_arg call? Does that need to change?

```
for(int i = 0; i < count; i++)
result += va_arg(args, int);
```

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

Why?

Example Format String Functions

Specifies number and types of arguments

Variable number of arguments

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

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]

Generally useful, but …

Format String Attacks

“*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

Assembly View

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

- For **variadic** functions, the program dynamically determines which registers and stack slots have arguments based upon a format specifier.

Example (1/3)

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

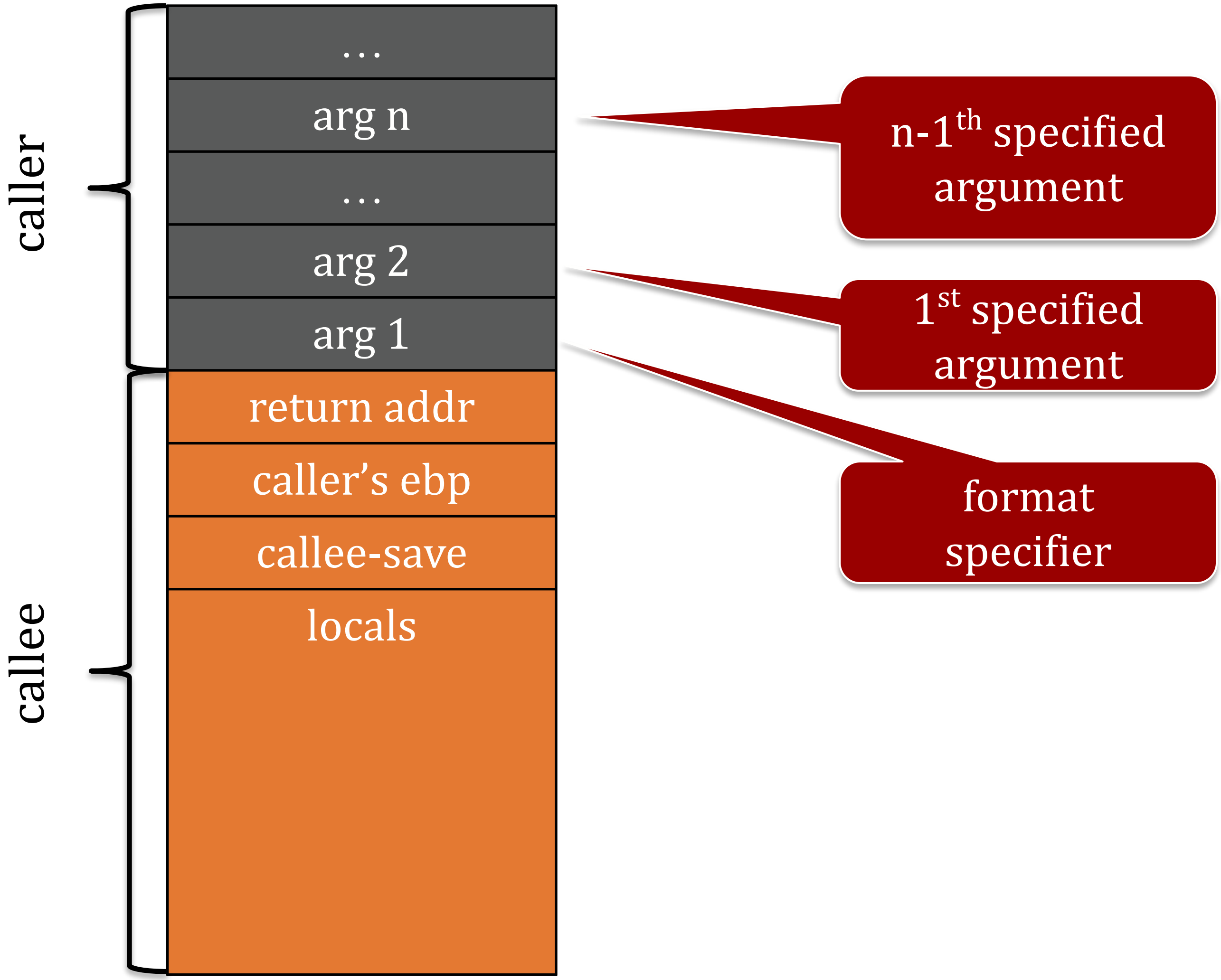
```
void debug(char *key, char *fmt, ...) {
  va_list ap;
  char buf[BUFSIZE];

  if (!KeyInList(key)) return;

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

Set up ap to point to stack using last fixed argument

Call vsprintf with args

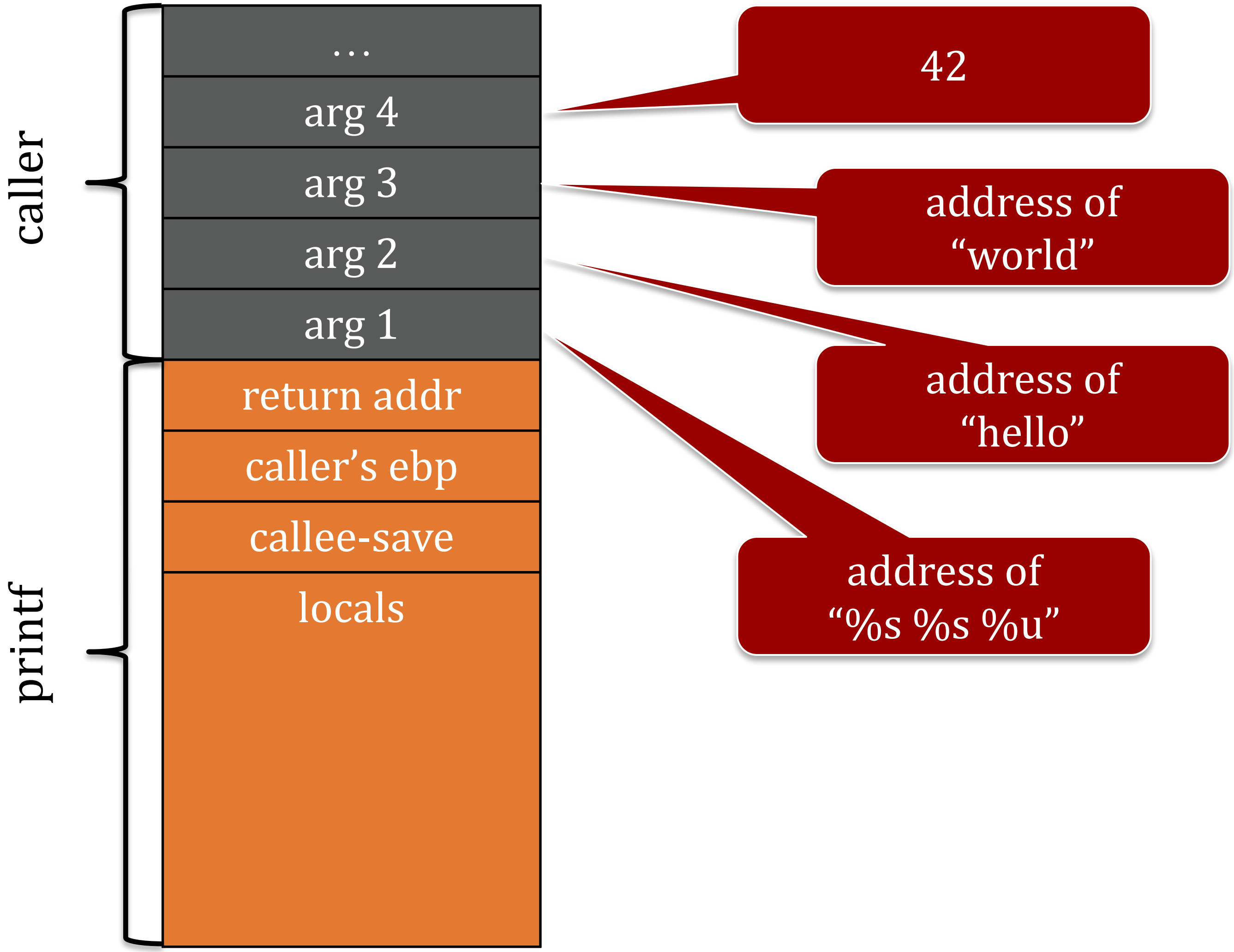
Cleanup

Stack Diagram



- 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 (2/3)



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

Example (3/3)

```
#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
```

Conversion Specifications

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

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

0 flag: zero-pad

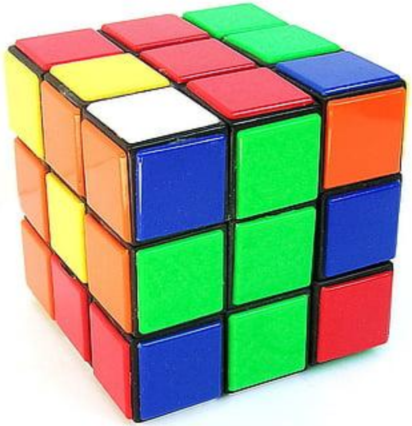
- `%08x`
  zero-padded 8-digit hexadecimal number

Minimum 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`

```
man -s 3 printf
```

Ένας Γρίφος



Μπορούμε να μαντεύουμε σωστά κάθε φορά;

```
int main(int argc, char ** argv) {
  int number, guess;
  srand(time(0));
  number = rand(); // Generates a random number between 0 and RAND_MAX
  if (argc > 1) printf(argv[1]);
  printf("\nGuess the number: "); fflush(stdout);
  scanf("%d", &guess);
  if (guess == number) {
    printf("Congratulations! You guessed the number in one shot. HOW?\n");
    return 0;
  }
```

Snippet from https://github.com/ethan42/fmt0/blob/master/guessme.c

Capability #1: Using a format string vulnerability we were able to *exfiltrate* data. Data from the stack that we were not supposed to have access to.

Direct Parameter Access Specifier - $
(It's a wonderful world out there!)

```
#include <stdio.h>


int main() {
  printf("Completed %1$d tasks (%1$d/%2$d total)\n", 8, 10);
}
```

What do you think the above program will print?

```
$ ./progress
Completed 8 tasks (8/10 total)
```

Can we make our previous solution shorter (better)?

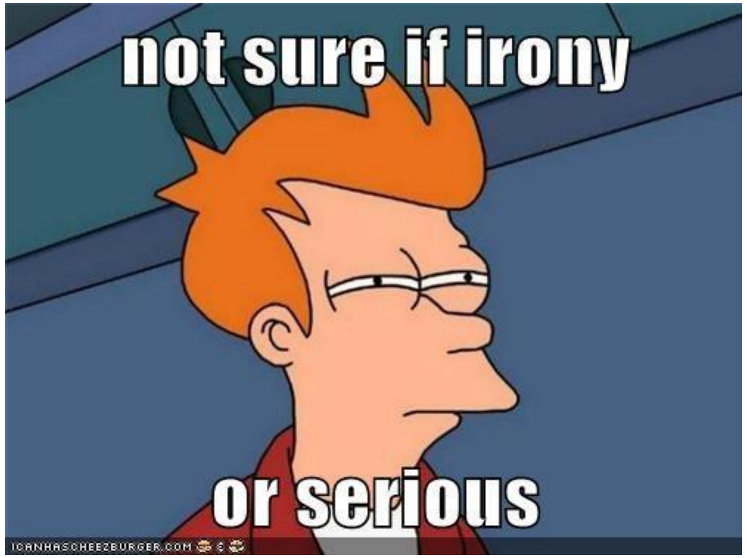
If stack data are unsafe because of stack walking, let's move everything important to the heap and be safe

Γρίφος #2: Μπορούμε να βρούμε το password;

```
int main(int argc, char ** argv) {
  char * secret = malloc(128);
  strcpy(secret, "my secure password");
  char guess[128];
  if (argc > 1) printf(argv[1]);
  printf("\npassword: "); fflush(stdout);
  fgets(guess, sizeof(guess), stdin);
  if (strncmp(secret, guess, strlen(secret)) == 0)
    printf("Access granted\n");
  else
    printf("Access denied\n");
}
```

Capability #1: Using a format string vulnerability we were able to *exfiltrate* data. Data from the stack **and where stack pointers point to** that we were not supposed to have access to.

OK, I guess they can read all our data, that's kinda bad :(. But at least we keep data integrity (they can't modify our data)



* %n enters the chat *

%n Format Specifier

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

```
int i;
printf("2002%n\n", (int *) &i);
printf("i = %d\n", i);
```

Output:

```
2002
i = 4
```

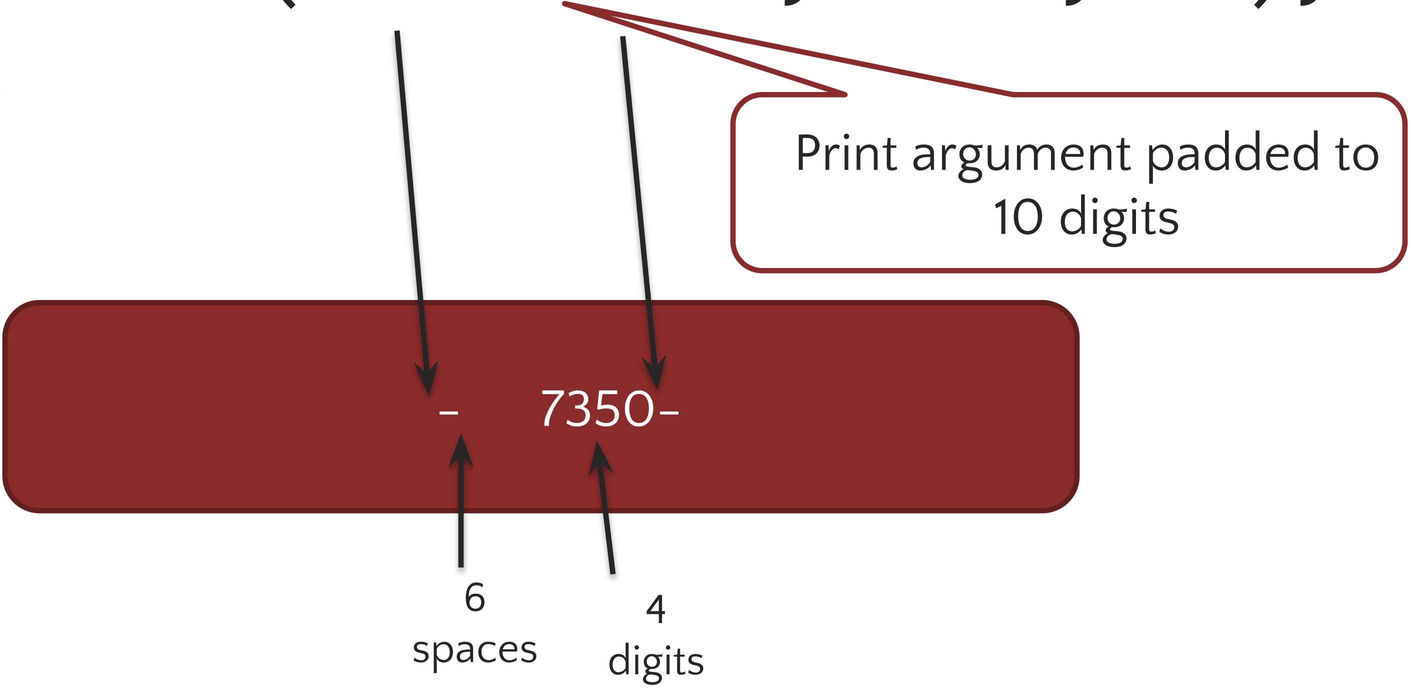
```
printf("%0*d", 5, 42);
=> 00042
```

Specifying Length

What does:

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

print?



Print argument padded to 10 digits

```
-      7350-
```

6 spaces

4 digits

Γρίφος #3: Μπορούμε να αλλάξουμε το password;

```
int main(int argc, char ** argv) {
  char * secret = malloc(128);
  strcpy(secret, "my secure password");
  char guess[128];
  if (argc > 1) printf(argv[1]);
  printf("\npassword: "); fflush(stdout);
  fgets(guess, sizeof(guess), stdin);
  if (strncmp(secret, guess, strlen(secret)) == 0)
    printf("Access granted\n");
  else
    printf("Access denied\n");
}
```

Probably got something like this

```
$ ./secret '%6578530c%39$n'
...snip...
password: bad
Access granted
```

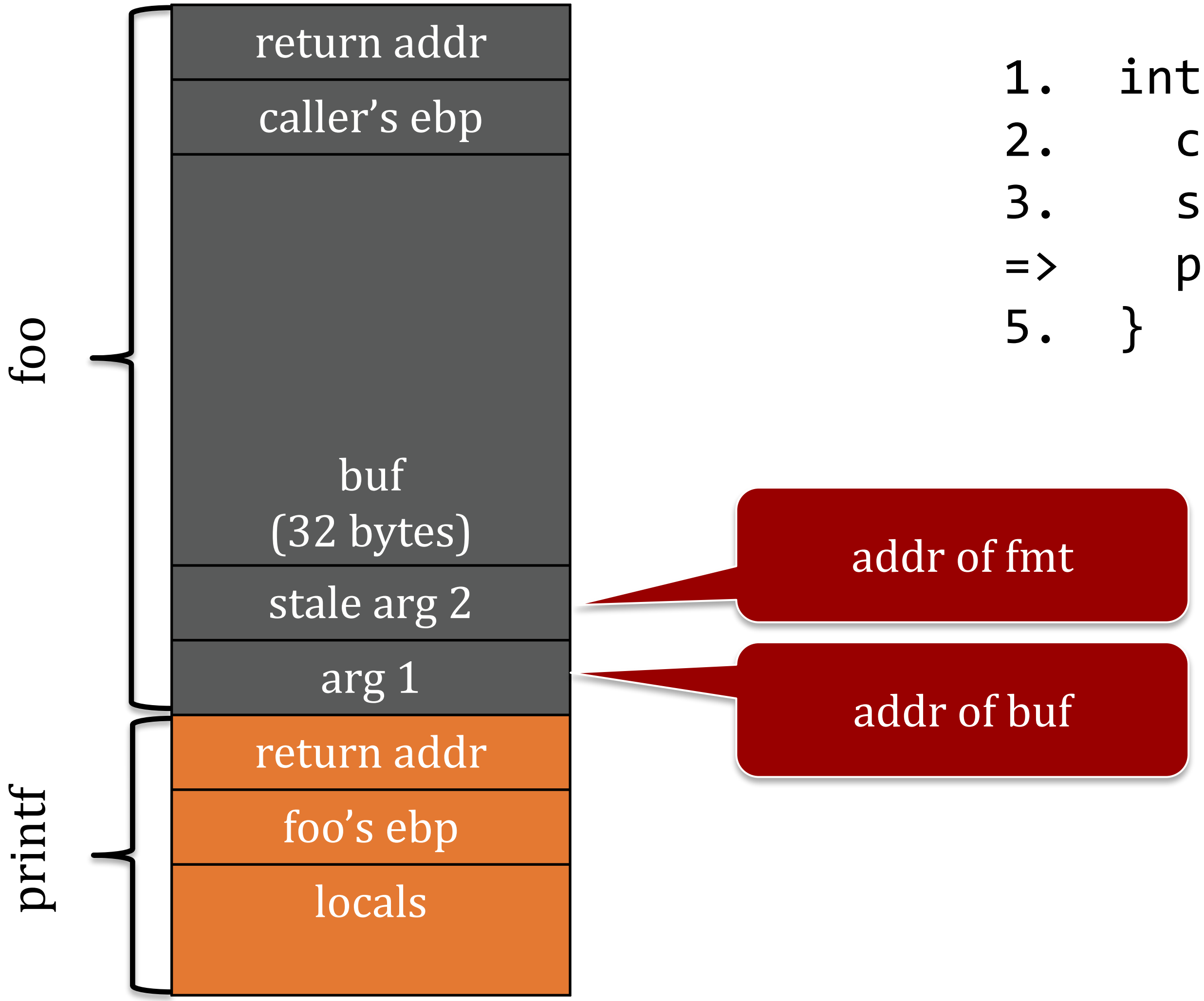
Capability #2: Using a format string vulnerability we we can write to any data pointed to from the stack.

A Toy Example

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

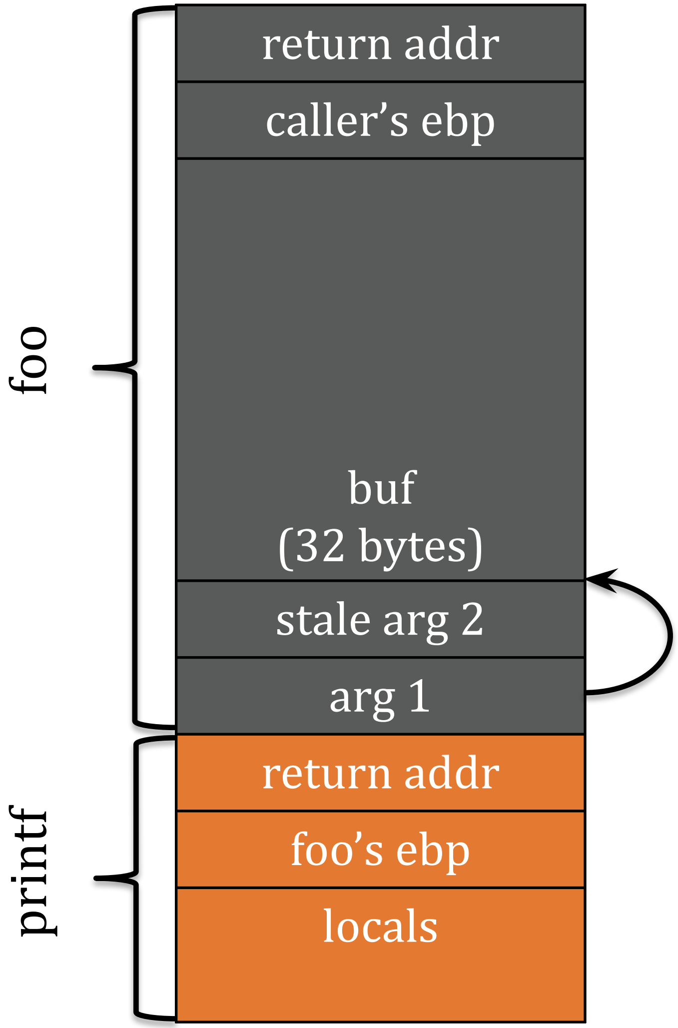
```
080483d4 <foo>:
 80483d4:    push   %ebp
 80483d5:    mov    %esp,%ebp
 80483d7:    sub    $0x28,%esp        ; allocate 40 bytes on stack
 80483da:    mov    0x8(%ebp),%eax    ; eax := M[ebp+8]  - addr of fmt
 80483dd:    mov    %eax,0x4(%esp)    ; M[esp+4] := eax  - push as arg 2
 80483e1:    lea    -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



foo

printf

return addr

caller’s ebp

buf
(32 bytes)

stale arg 2

arg 1

return addr

foo’s ebp

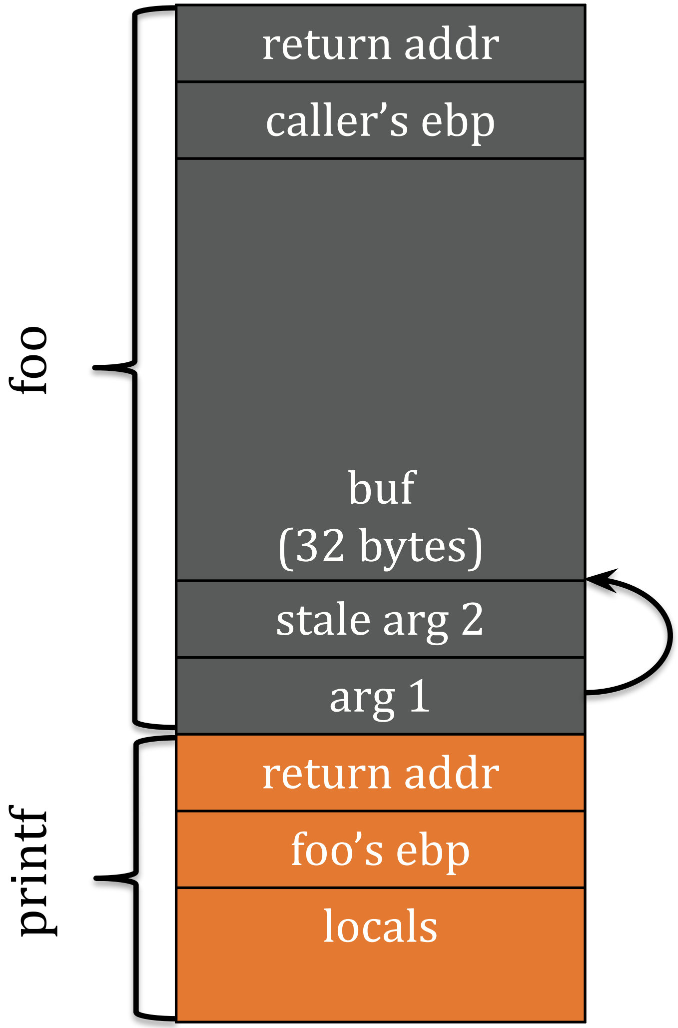
locals

```
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. $\underbrace{\%x\%x\ldots\%x}_{\text{11 times}}$

Viewing Specific Address—1

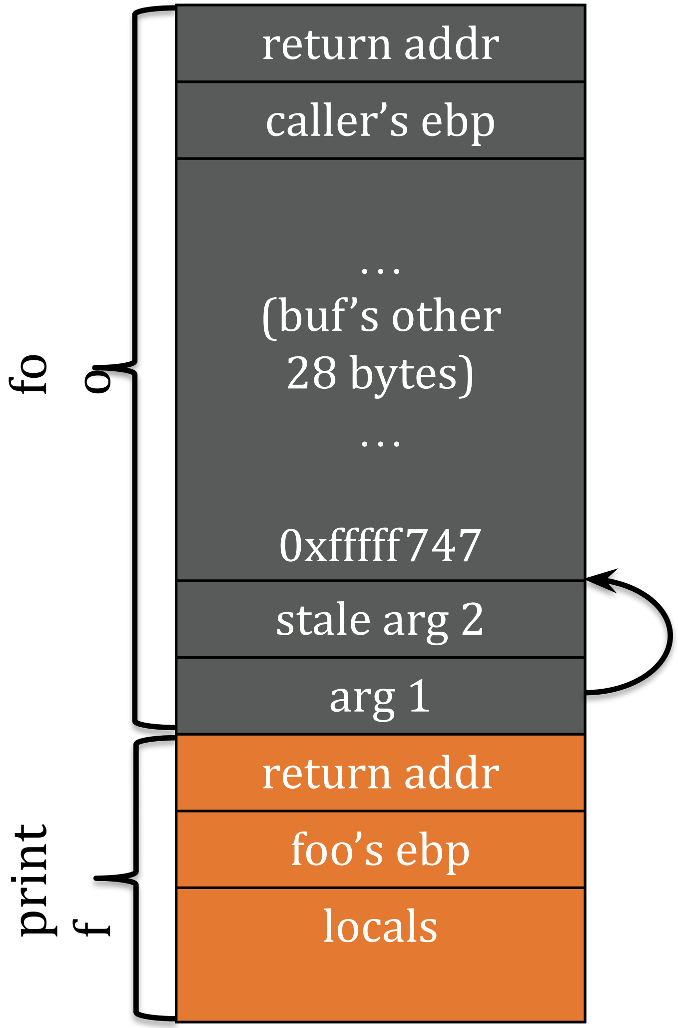


```
1.   int foo(char *fmt) {
2.     char buf[32];
3.     strcpy(buf, fmt);
=>     printf(buf);
5.   }
```

Observe: buf is ***above*** 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



foo

return addr

caller's ebp

…

(buf's other 28 bytes)

…

0xffff747

stale arg 2

arg 1

printf

return addr

foo's ebp

locals

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

Idea! Encode address to peek in buf first.

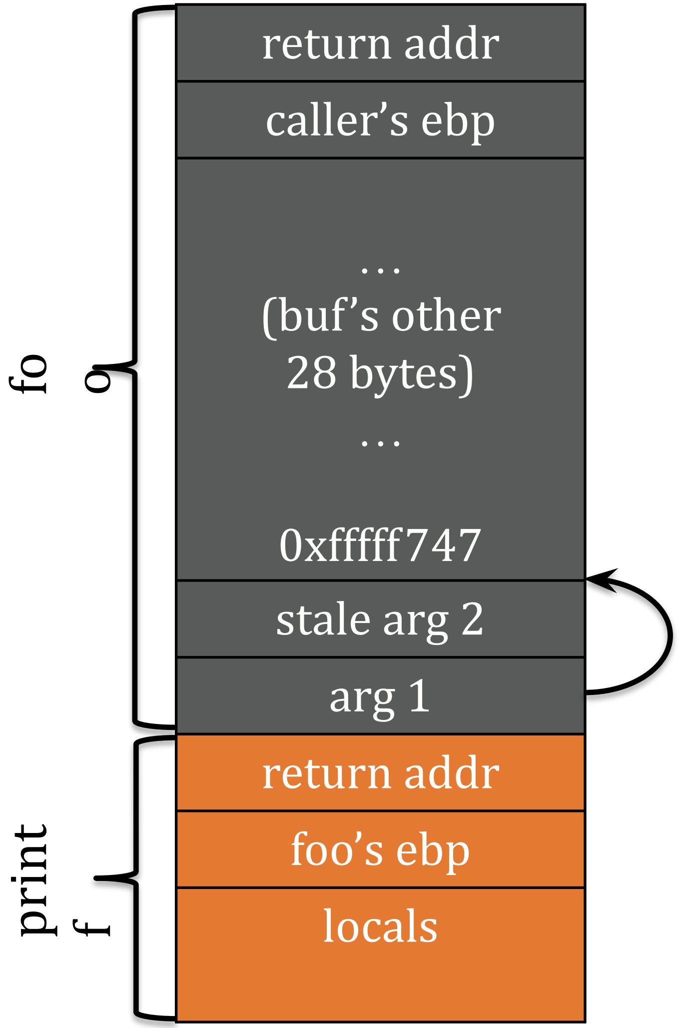
Address `0xffff747` is

`\x47\xf7\xff\xbf`

in *little endian*.

\x47\xf7\xff\xff%x%s

Writing to Specific Address


foo
return addr
caller's ebp
…
(buf's other 28 bytes)
…
0xfffff747
stale arg 2
arg 1
printf
return addr
foo's ebp
locals

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

Same Idea! Encode address to peek in buf first. Address `0xfffff747` is

`\x47\xf7\xff\xbf`

in *little endian*.

`\x47\xf7\xff\xff`%x%n

Wait! If you could write to any memory region, which one would you choose?

The instruction pointer (RIP) is your friend :D

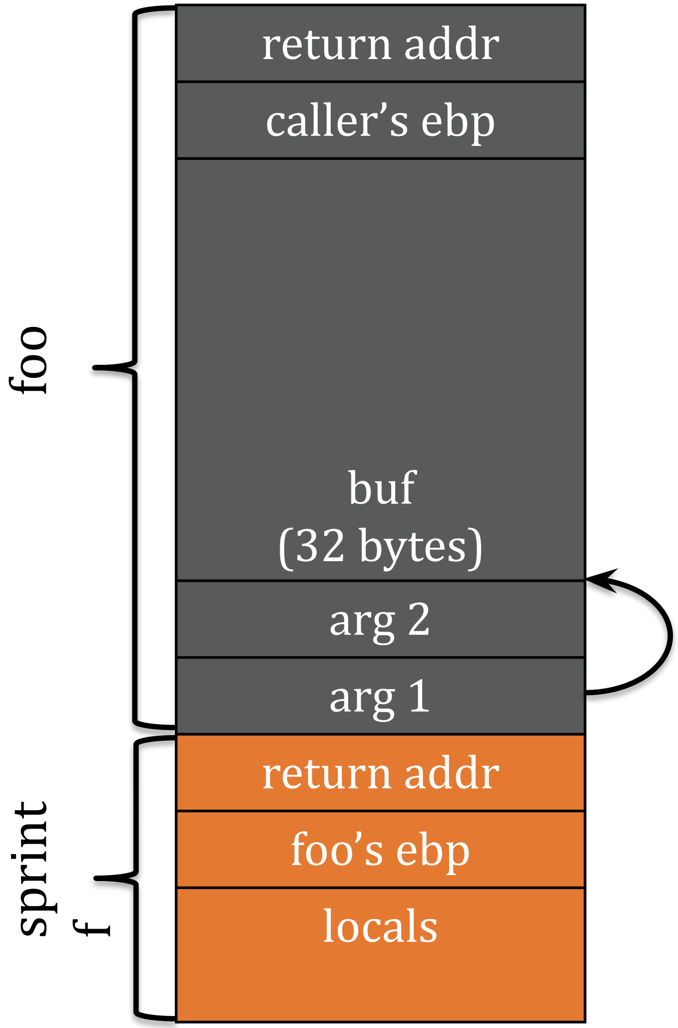
Capability #2: Using a format string vulnerability we may be able to write anything anywhere (aka *write-what-where* exploit), which typically translates to arbitrary control of execution

Format Strings: a type of Control Flow Hijack

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

- Overwrite a function pointer or similar structure that may get invoked during execution (GOT, destructors, exception handlers and more).

Overflow by Format String



foo

return addr

caller’s ebp

buf
(32 bytes)

arg 2

arg 1

sprintf

return addr

foo’s ebp

locals

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

Overwrite
return address

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

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

Shellcode with
nop slide

Ευχαριστώ και καλή μέρα εύχομαι!

Keep hacking!