CS 88: Security and Privacy

05: Software Security – Stack Buffer Overflow, Integer Overflow and Format String Attacks

02-06-2024

SWARTHMORE COLLEGE

Announcements

- lab checkpoint is due today
- please come by for ninja office hours 4-5pm!

Reading Quiz

Today

- Software attacks
    - Integer Overflow Attacks
    - Format String Attacks
    - Return Oriented Programming

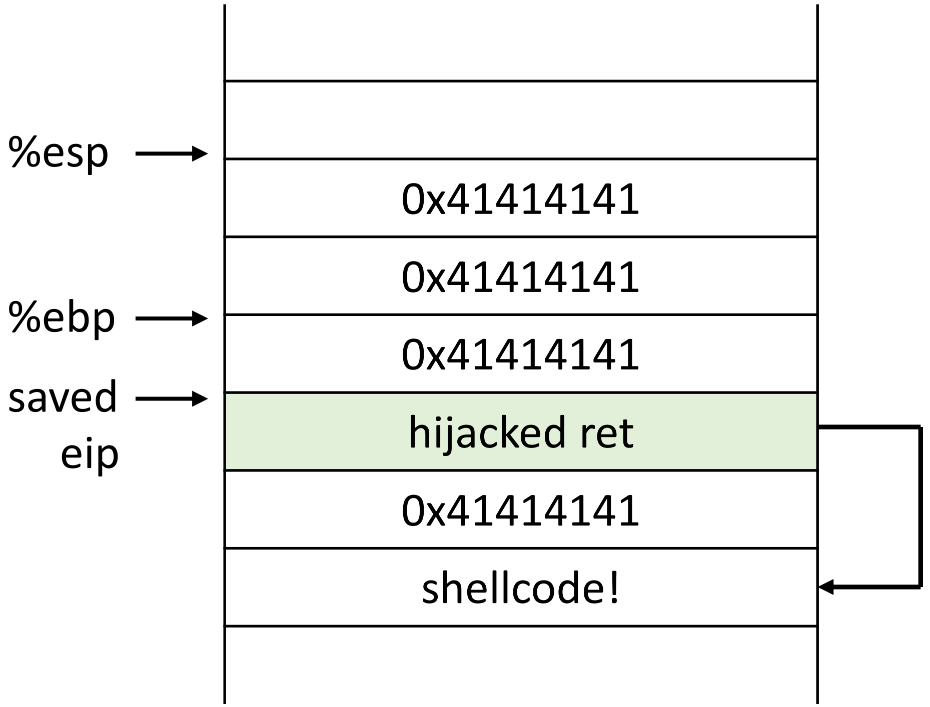
Buffer Overflows

Buffer Overflows

- An anomaly that occurs when a program writes/reads data beyond the boundary of a buffer
- Canonical software vulnerability
  - ubiquitous in system software
  - OSes, web servers, web browsers
- If your program crashes with memory faults, you probably have a buffer overflow vulnerability

Better Hijacking Control

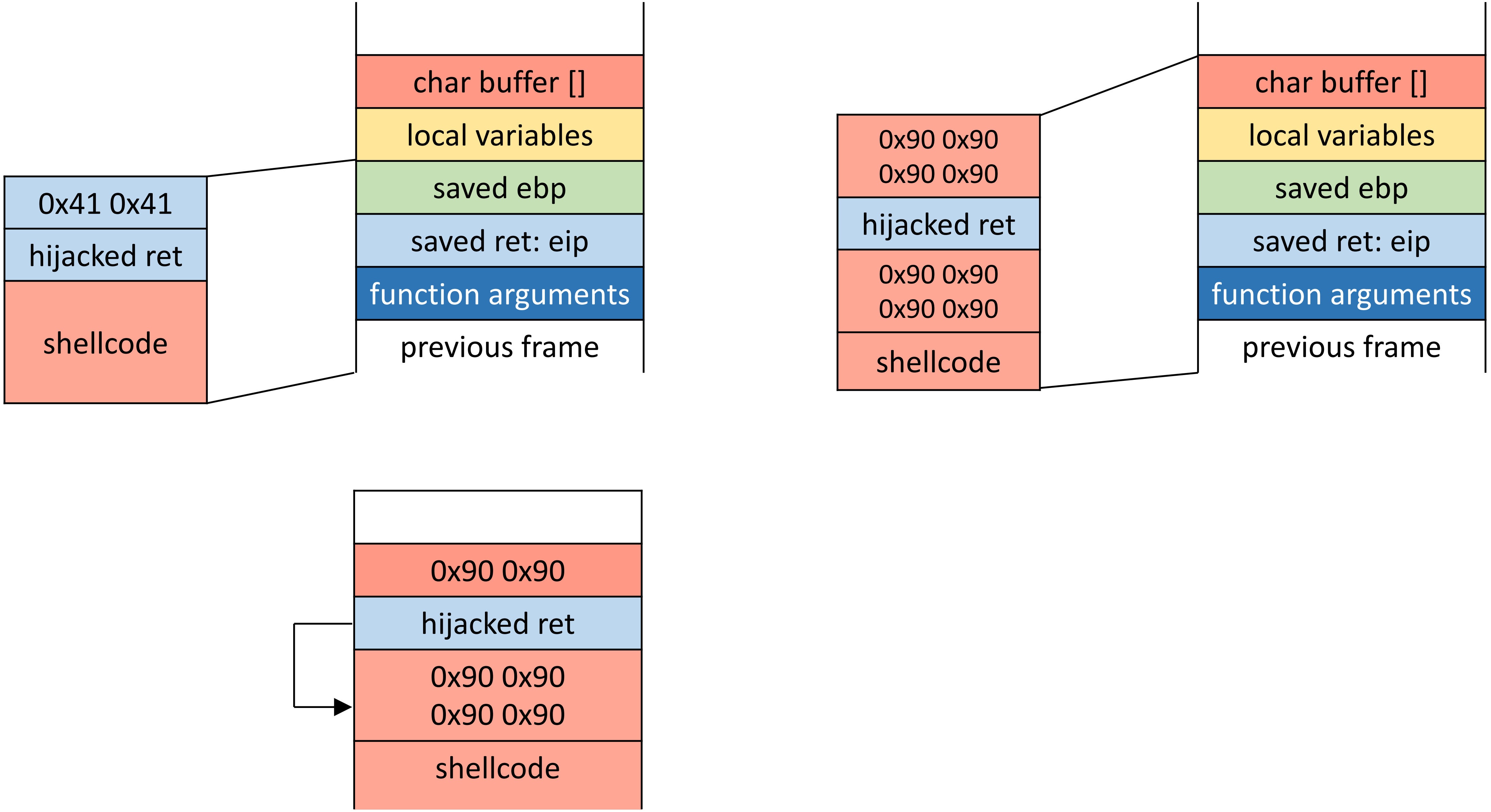
```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xfoo5ball ;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



Jump to attacker supplied code where?

- put code in the string
- jump to start of the string

Putting it all together



Some Unsafe C lib Functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)

printf (conts char *format, …)

⋮

Avoid strcpy, …

- We have seen that `strcpy` is unsafe
  - `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until “`\0`” is encountered, ignoring the size of `buf`
  - Avoid `strcpy()`, `strcat()`, `gets()`, etc.
  - Use `strncpy()`, `strncat()`, instead
- Even these are not perfect… (e.g., no null termination)
- Always a good idea to do your own validation when obtaining input from untrusted source
- Still need to be careful when copying multiple inputs into a buffer

Cause of vulnerability: No Range 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`

Width Overflows

```
uint32_t x = 0x10000;
uint16_t y = 1;
uint16_t z = x + y;      // z = ?
```

- Width overflows occur when assignments are made to variables that can't store the result
- Integer promotion
  - Computation involving two variables *x, y* where width(*x*) > width(*y*)
  - *y* is promoted such that width(*x*) = width(*y*)

Sign Overflows

```
int f(char* buf, int len) {
    char dst_buf[64];
    if (len > 64)
        return 1;
    memcpy(dst_buf, buf, len);
    return 0;
}
```

```
memcpy(void *, void *, unsigned int)
```

- Sign overflows occur when an unsigned variable is treated as signed, or vice-versa
  - Can occur when mixing signed and unsigned variables in an expression
  - Or, wraparound when performing arithmetic

Broward Vote-Counting Blunder Changes Amendment Result

POSTED: 1:34 pm EST November 4, 2004

BROWARD COUNTY, Fla. -- The Broward County Elections Department has egg on its face today after a computer glitch misreported a key amendment race, according to WPLG-TV in Miami.

Amendment 4, which would allow Miami-Dade and Broward counties to hold a future election to decide if slot machines should be allowed at racetracks, was thought to be tied. But now that a computer glitch for machines counting absentee ballots has been exposed, it turns out the amendment passed.

Broward County Mayor Ilene Lieberman says voting counting error is an "embarrassing mistake."

"The software is not geared to count more than 32,000 votes in a precinct. So what happens when it gets to 32,000 is the software starts counting backward," said Broward County Mayor Ilene Lieberman.

That means that Amendment 4 passed in Broward County by more than 240,000 votes rather than the 166,000-vote margin reported Wednesday night. That increase changes the overall statewide results in what had been a neck-and-neck race, one for which recounts had been going on today. But with news of Broward's error, it's clear amendment 4 passed.

Heartbleed vulnerability

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    uchar payload [HeartbeatMessage.payload_length];
    uchar padding[padding_length];
  } HeartbeatMessage;
```

If your program has a buffer overflow bug, you should assume that the bug is exploitable and an attacker can take control of your program.

Other overflow targets

- Format strings in C
- Return Oriented Programming

Format String Vulnerabilities

Variable arguments in C

In C, we can define a function with a variable number of arguments

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

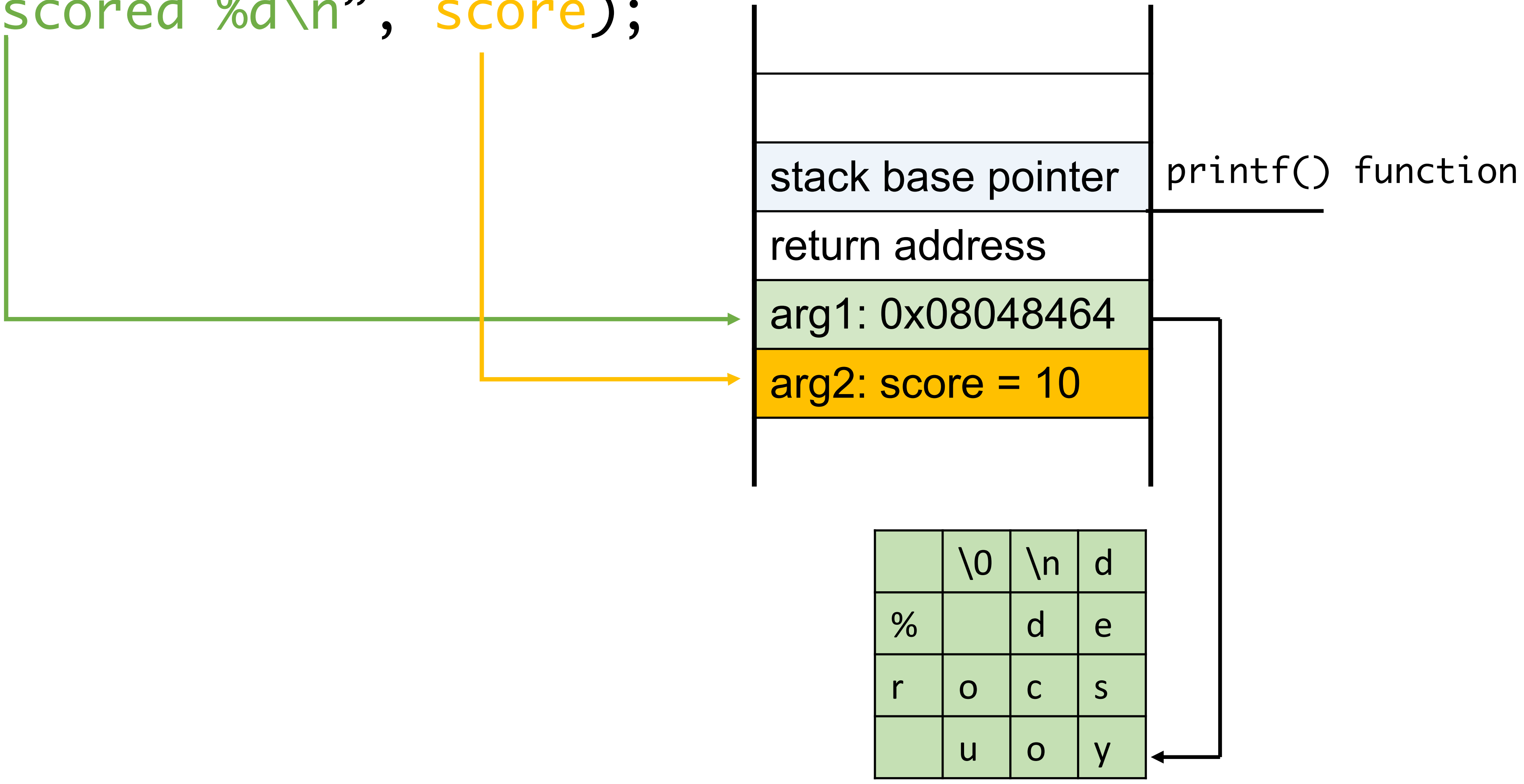
Usage:

```
printf("hello world");
printf("length of %s = %d \n", str, str.length());
```

format specification encoded by special % characters

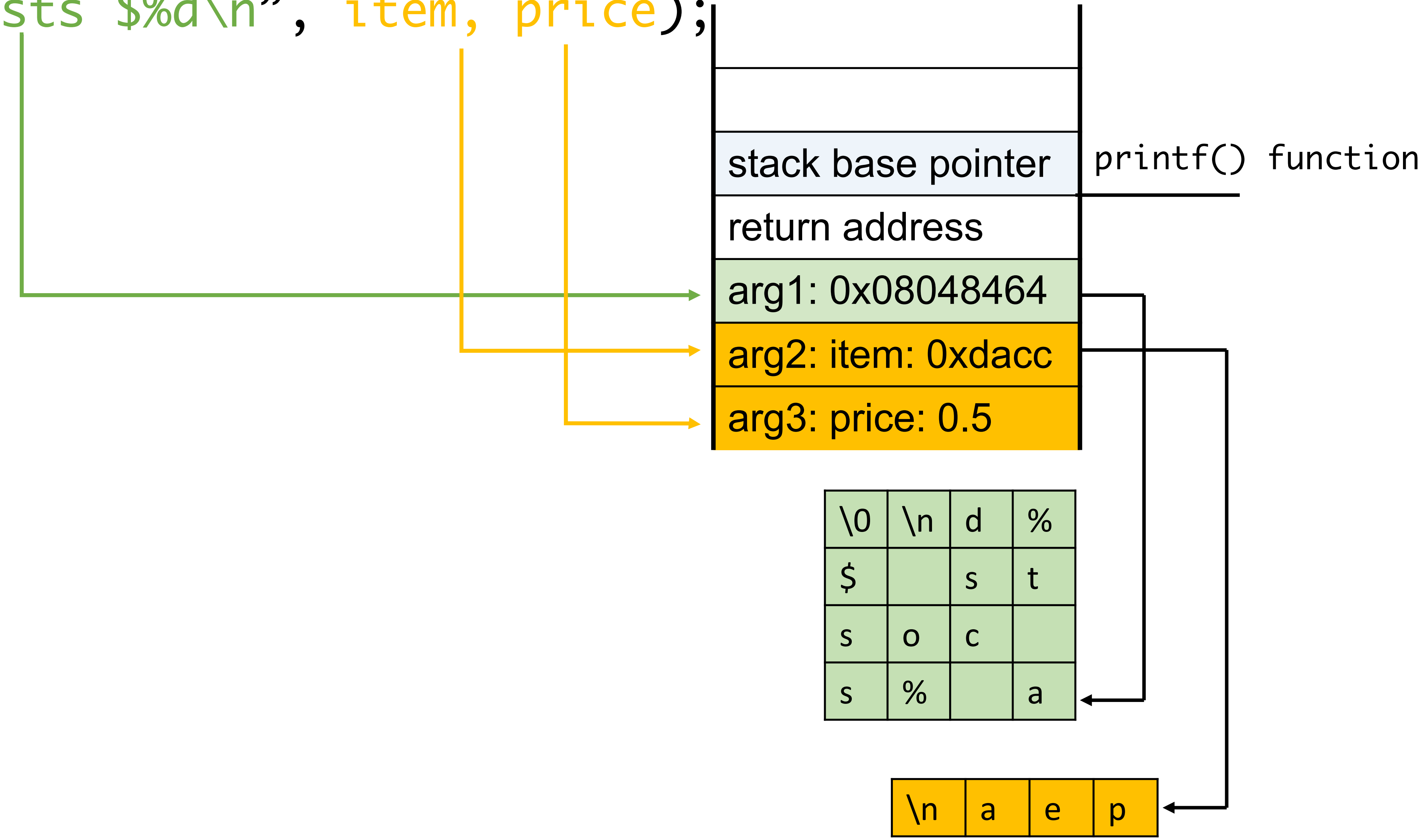
fun with format strings

```
printf("you scored %d\n", score);
```



fun with format strings

```
printf("a %s costs $%d\n", item, price);
```



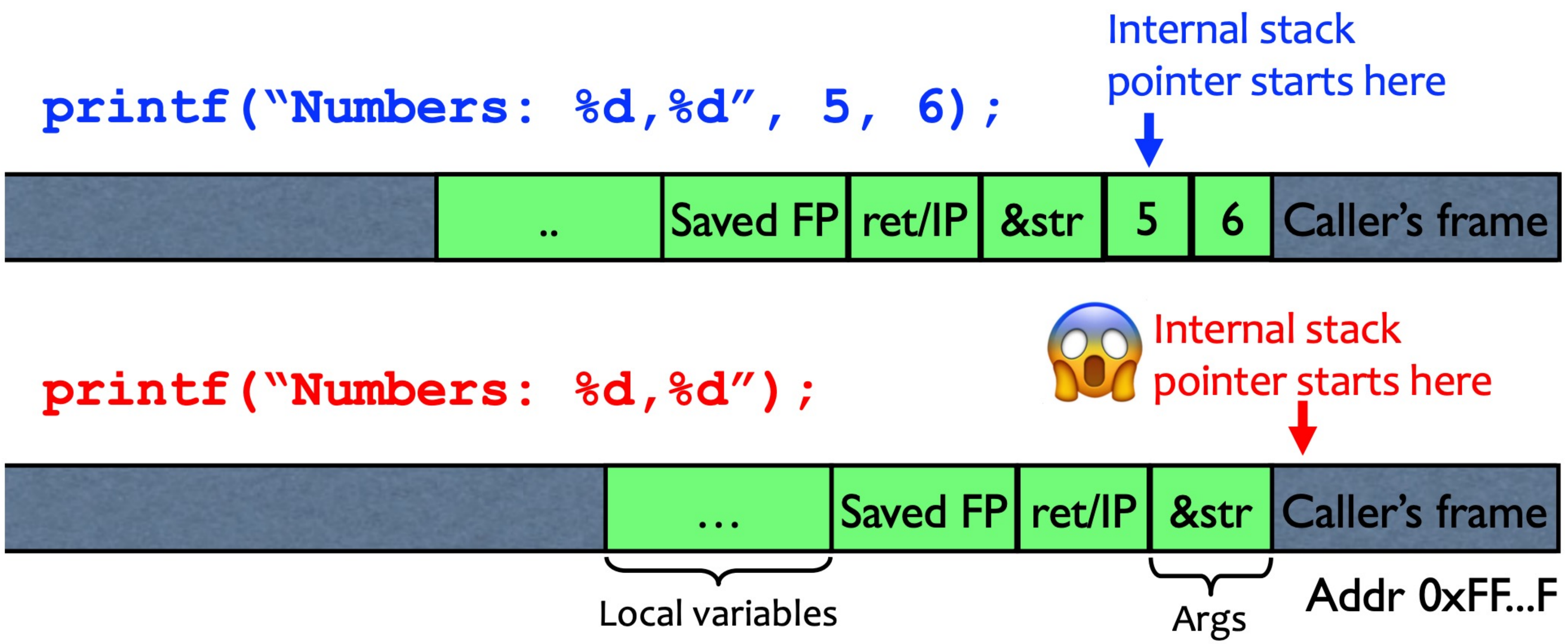
Implementation of printf

- 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 */
}
```

printf has an internal stack pointer

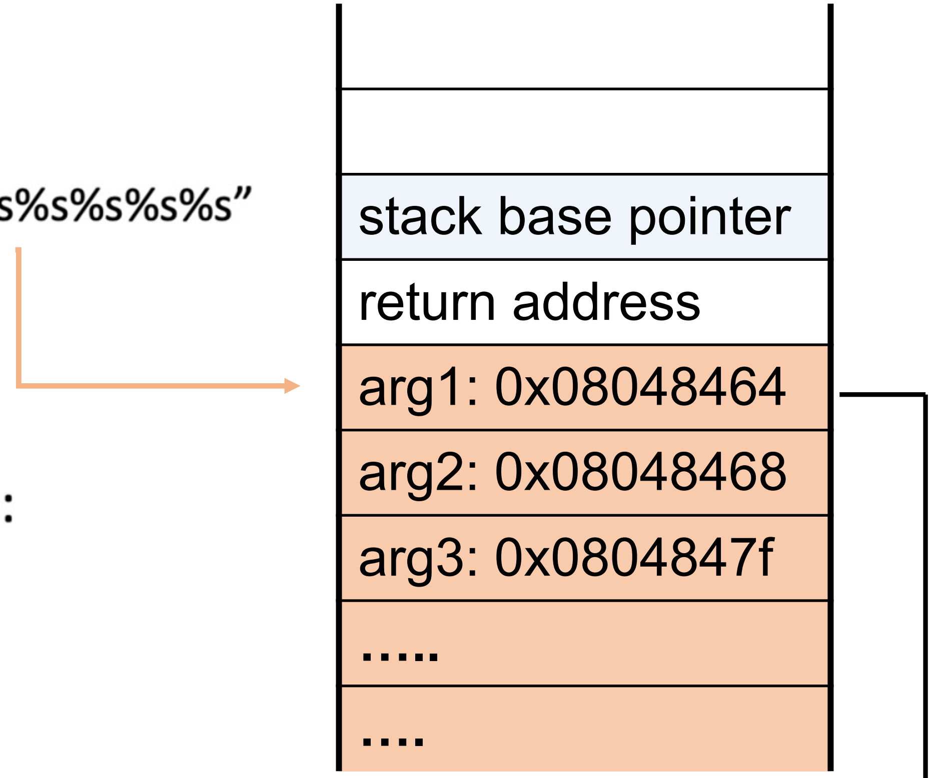
Closer look at the stack



Sloppy use of printf

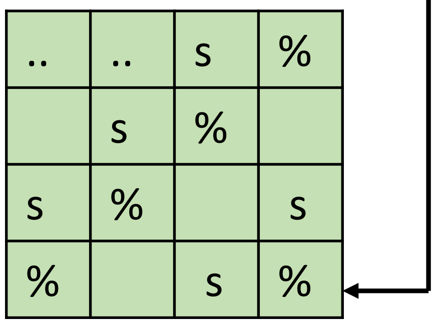
```
void main(int argc, char* argv[])
{
      printf( argv[1] );
}
```

argv[1] = “%s%s%s%s%s%s%s%s%s%s%s”



Attacker controls format string gives all sorts of control:

- Print stack contents
- Print arbitrary memory
- Write to arbitrary memory



Format specification encoded by special % characters

Format Specifiers

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address
- `printf (“overflow this!%n”, &myVar);`
  - Writes 14 into `myVar`.

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address
- `printf ("overflow this!%n", &myVar);`
  - Writes 14 into `myVar`.
- What if printf does not have an argument?
  - `char buf[16] = "Overflow this!%n";`
  - `printf(buf);`

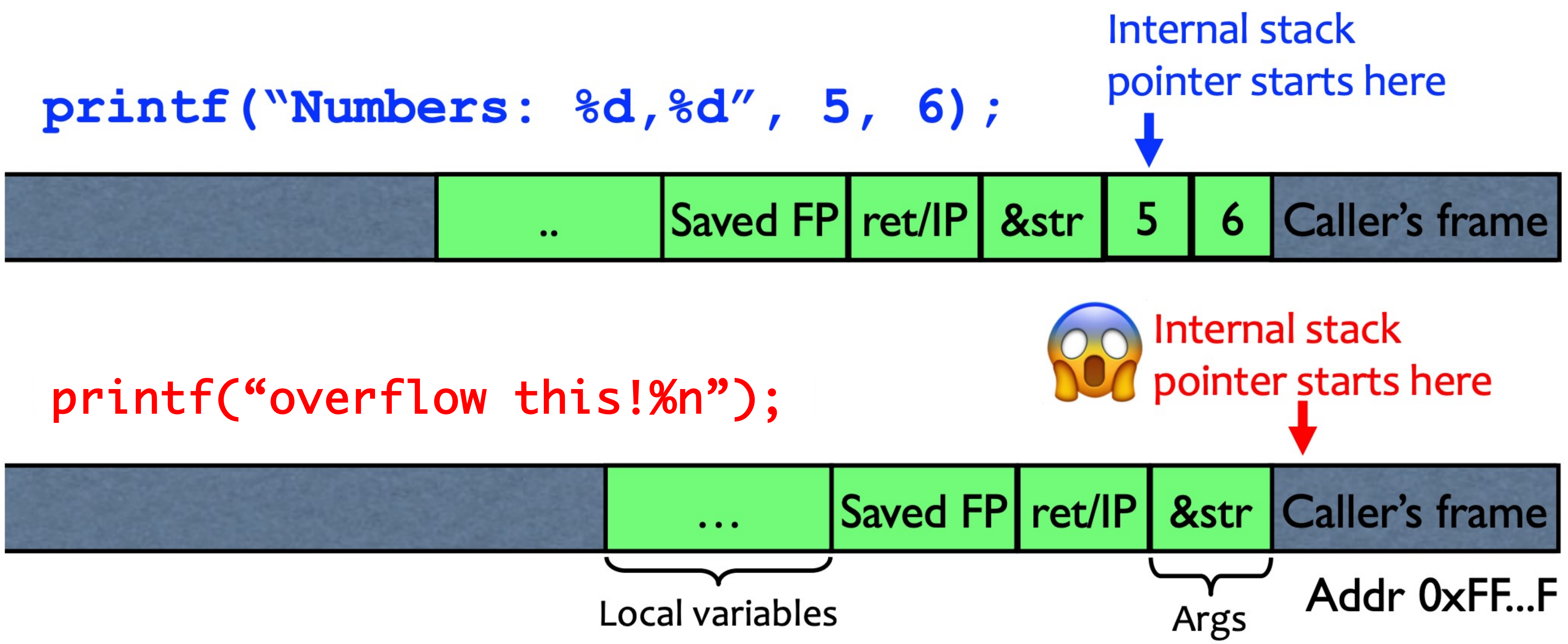
A. Store the value 14 in buf
B. Store the value 14 on the stack (specify where)
C. Replace the string Overflow with 14
D. Something else

The %n format specifier

- `%n` format symbol tells `printf` to write the number of characters that have been printed
  - Argument of `printf` is interpreted as a destination address
- `printf (“overflow this!%n”, &myVar);`
  - Writes 14 into `myVar`.
- What if printf does not have an argument?
  - `char buf[16] = “Overflow this!%n”;`
  - `printf(buf);`

- Stack location pointed to by printf’s internal stack pointer will be interpreted as an address
- Write # characters at this address

Closer look at the stack



Write 14 into the caller's frame!

fun with printf: what’s the output of the following statements?

```
printf(“100% dive into C!”)

printf(“100% samy worm”);

printf(“%d %d %d %d”);

printf(“%d %s);

printf(“100% not another segfault!”);
```

fun with printf: what's the output of the following statements?

```
printf("100%dive into C!")
```

100 + value 4 bytes below retaddress as an integer + "ive"

```
printf("100%samy worm");
```

prints bytes pointed to by the stack entry up through the first NULL

```
printf("%d %d %d %d");
```

print series of stack entries as integers

```
printf("%d %s);
```

print value 4 bytes below return address plus bytes pointed to by the preceding stack entry

```
printf("100% not another segfault!");
```

prints 100 not another segfault! and stores the number 3 on the stack

Viewing the stack

We can show some parts of the stack memory by using a format string like this:

C code

```
printf ("%08x.%08x.%08x.%08x.%08x\n");
```

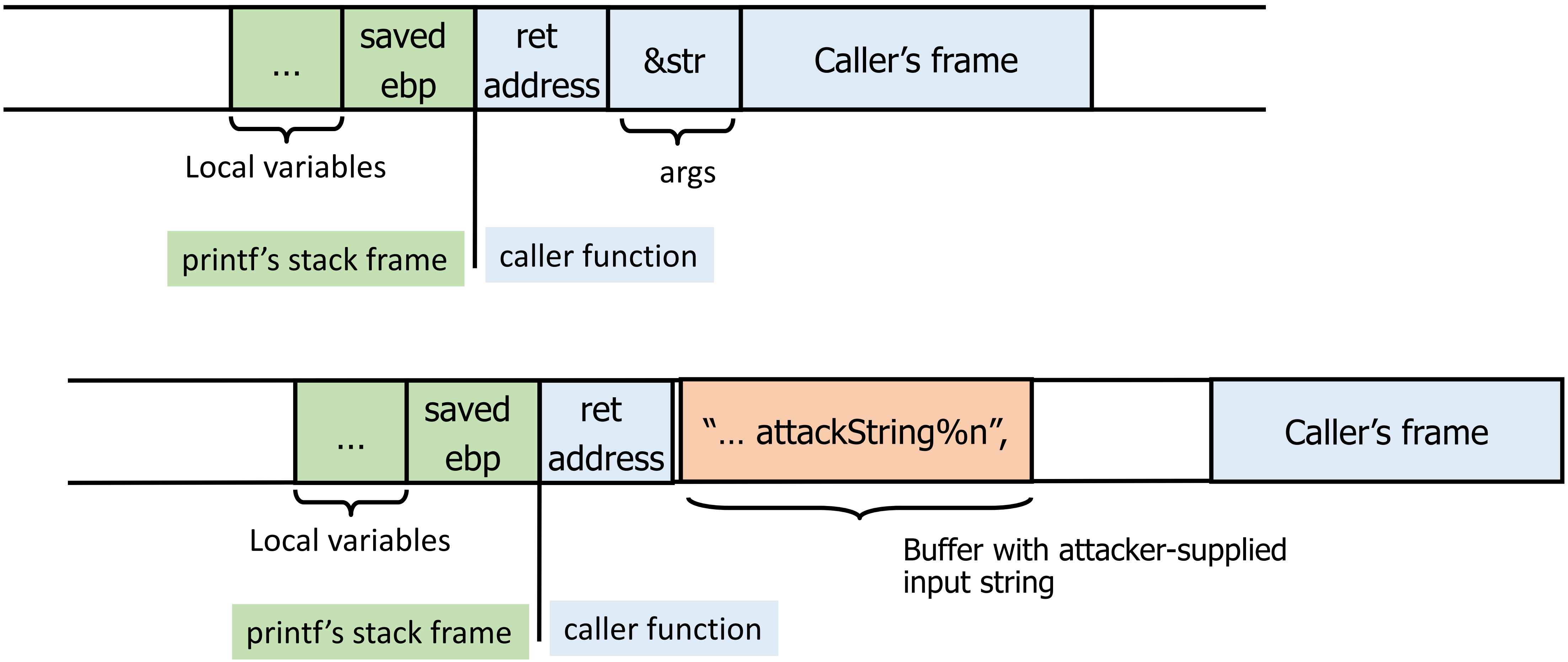
Output

```
40012980.080628c4.bffff7a4.00000005.08059c04
```

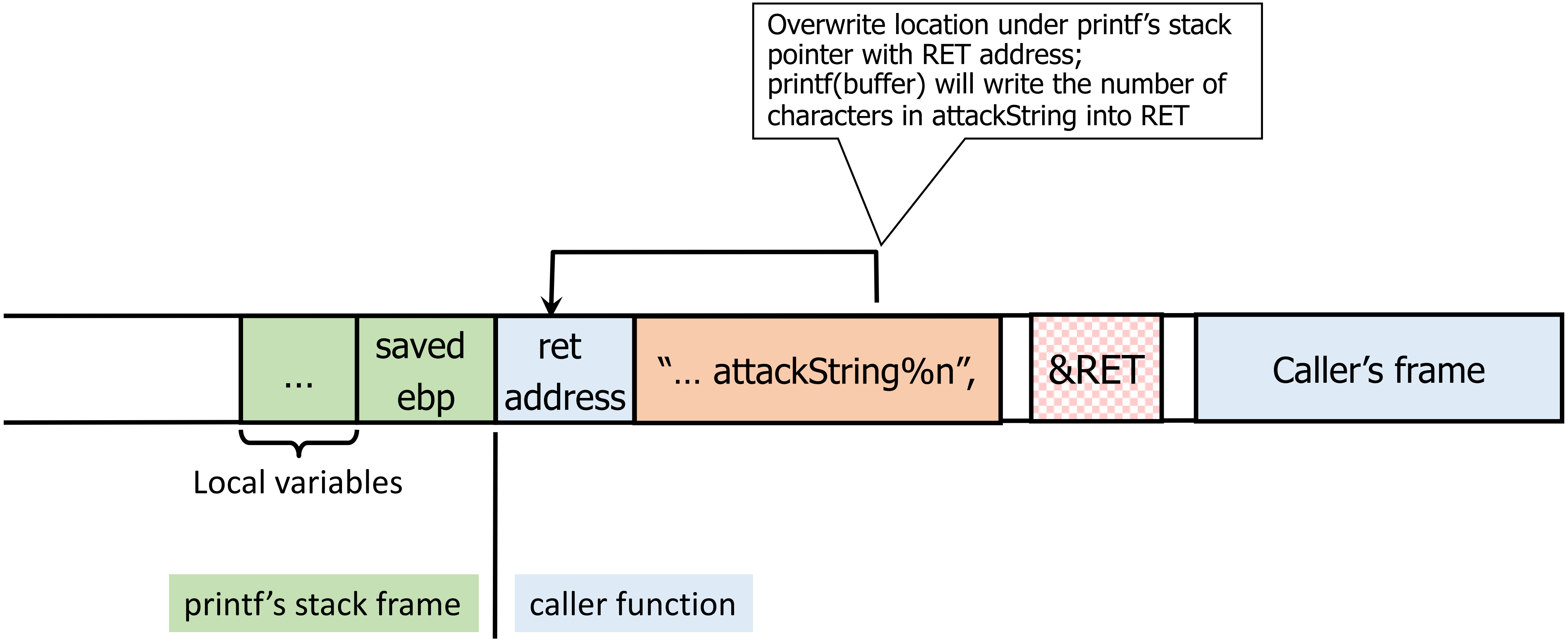
instruct printf:

- retrieve 5 parameters
- display them as 8-digit padded hexademical numbers

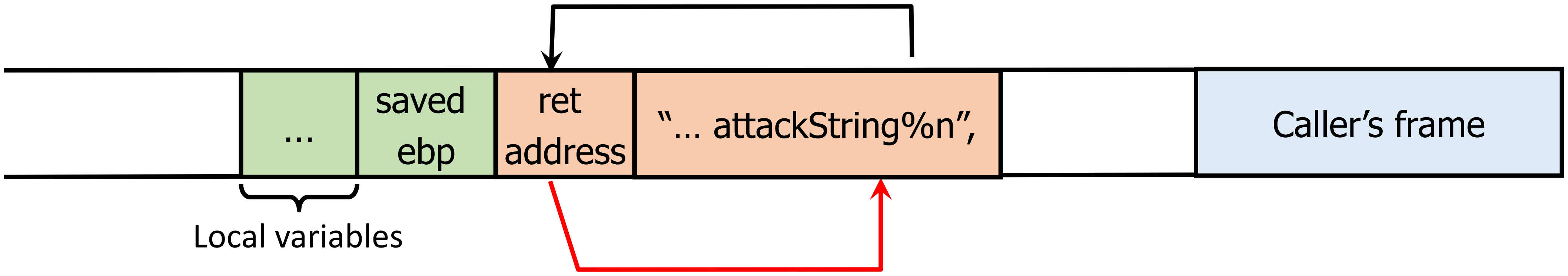
Using %n to Mung Return Address



Using %n to Mung Return Address



Using %n to Mung Return Address



C has a concise way of printing multiple symbols:

- %Mx will print exactly 4M bytes (taking them from the stack).
- Attack string should contain enough "%Mx" so that the number of characters printed is equal to the most significant byte of the address of the attack code.
- Repeat three times (four "%n" in total) to write into &RET+1, &RET+2, &RET+3, thus replacing RET with the address of attack code byte by byte.

See "Exploiting Format String Vulnerabilities" for details

If your program has a format string bug, assume that the attacker can learn all secrets stored in memory, and assume that the attacker can take control of your program.

Secure coding guidelines

1. Only use the memory allocated from a call to `malloc`. Do not access/ensure no access to memory that is out of bounds.
2. `Free` dynamically allocated memory exactly once.
3. Never access freed memory.
4. Always check the return value from a call to `malloc` (is NULL?).
5. After every call to `free`, re-assign the pointer to `NULL`.
6. Zero out sensitive data before freeing it using `memset`.
7. Do not make any assumptions regarding the memory addresses returned from `malloc`.

https://github.com/shellphish/how2heap

Source: https://heap-exploitation.dhavalkapil.com/attacks/

Buffer Overflow: Causes

- Typical 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
  - Overwrite saved EIP, function callback pointer, etc.

Integer overflows

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

int main(int argc, char *argv[]){
	unsigned short s;
	int i;
	char buf[80];

	if(argc < 3){
		return -1;
	}
```

```
	i = atoi(argv[1]);
	s = i;

	if(s >= 80) {		/* [w1] */
		printf("Oh no you don't!\n");
		return -1;
	}

	printf("s = %d\n", s);

	memcpy(buf, argv[2], i);
	buf[i] = '\0';
	printf("%s\n", buf);

	return 0;
}
```

Output

```
$ ./overflow 5 hello
s = 5
hello

$ ./overflow 80 hello
Oh no you don't

$ ./overflow 65536 hello
s = 0
Segmentation fault (core dumped)
```

What’s wrong with this code?

```
#define BUF_SIZE 16
char buf[BUF_SIZE];
void vulnerable()
{
    int len = read_int_from_network();
    char *p = read_string_from_network();
    if(len > BUF_SIZE) {
        printf("Too large\n");
        return;
    }
    memcpy(buf, p, len);
}
```

Integer overflow.
len of type int
memcpy takes an unsigned int

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

Off-By-One Overflow

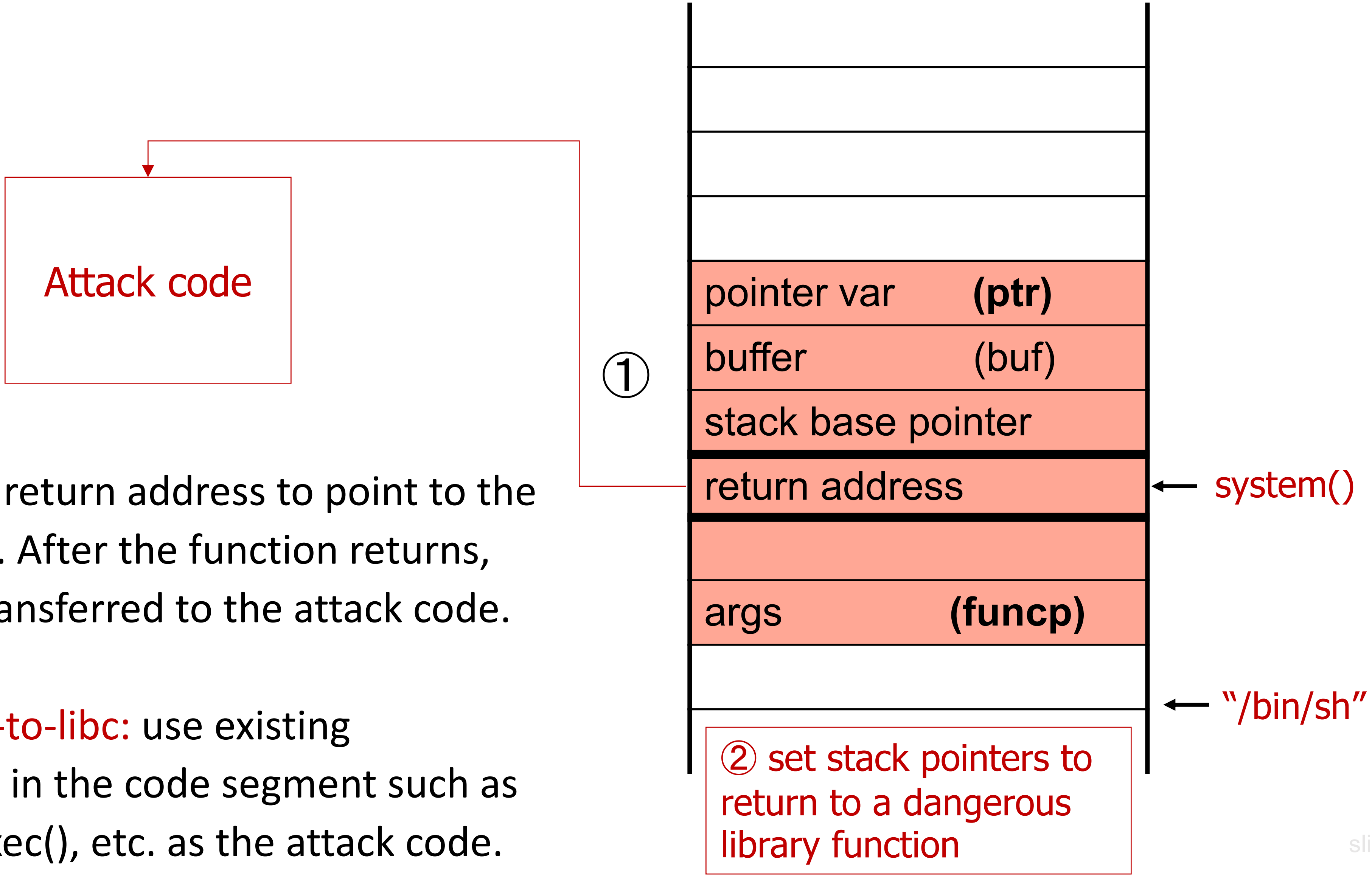
Home-brewed range-checking string copy

```
void notSoSafeCopy(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)
        notSoSafeCopy(argv[1]);
}
```

This will copy **513** characters into buffer. Oops!

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

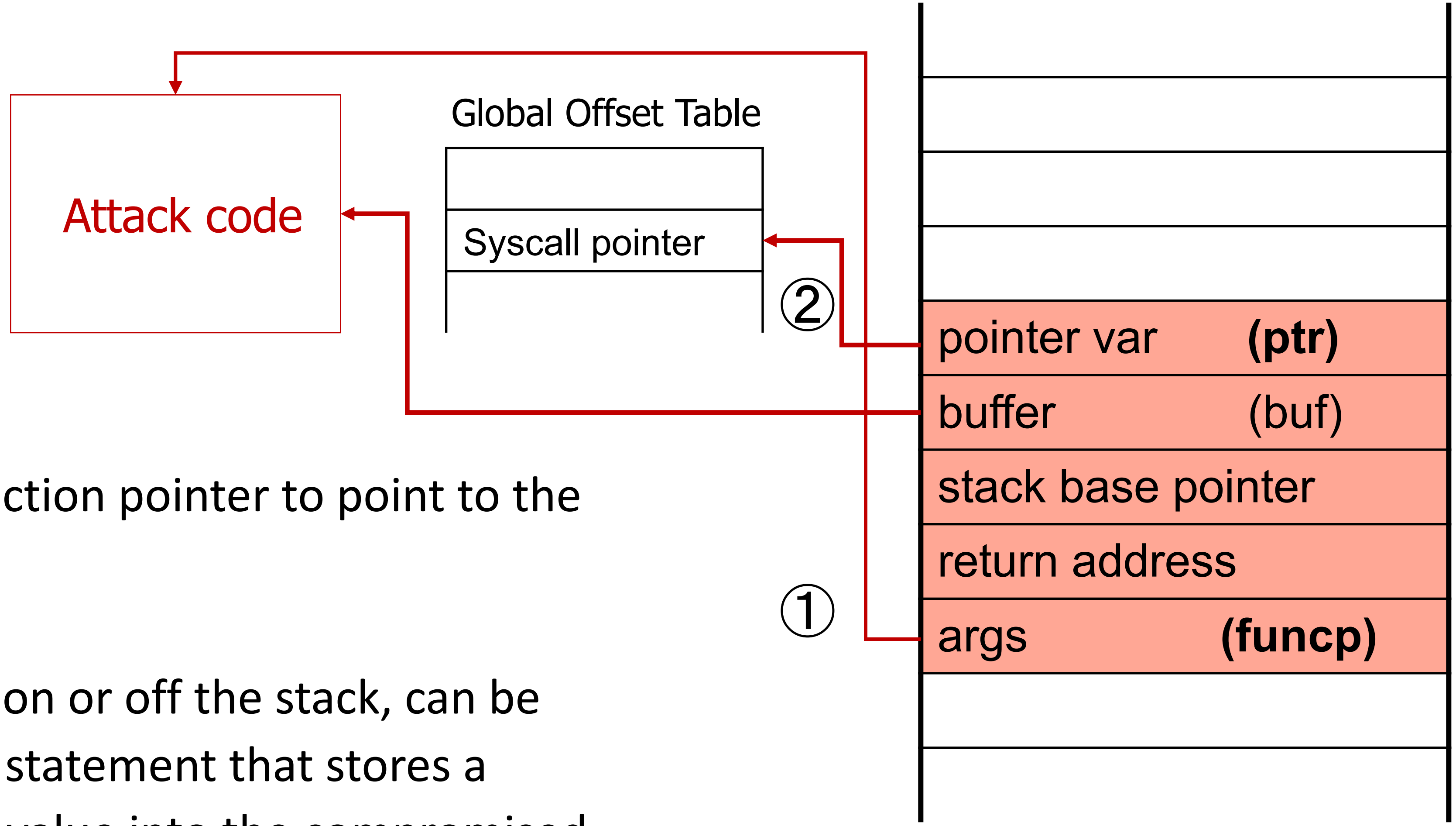
Other Control Hijacking Opportunities: return-to-libc attack



(1) Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

(2) … or return-to-libc: use existing instructions in the code segment such as system(), exec(), etc. as the attack code.

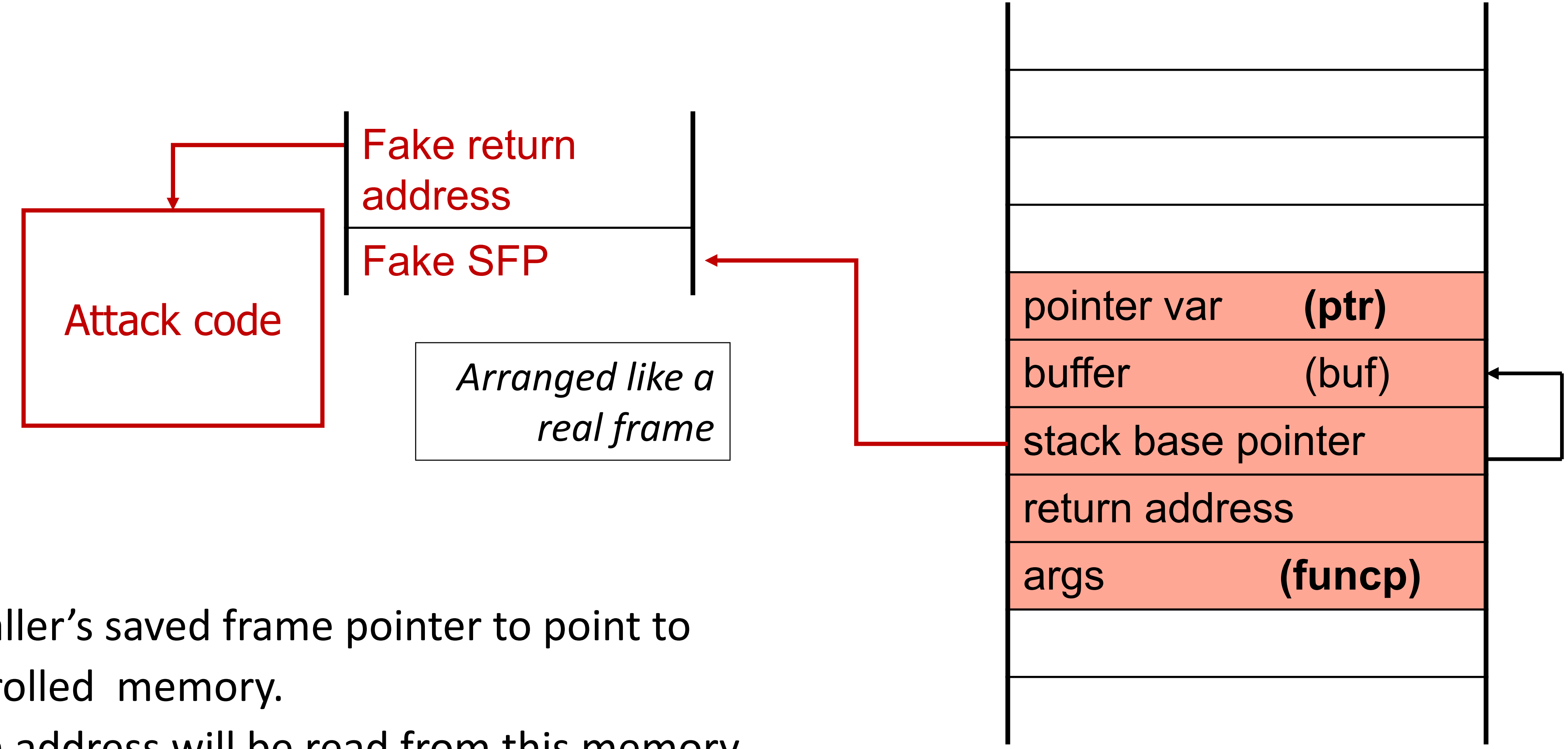
Other Control Hijacking Opportunities: Function Pointers



(1) Change a function pointer to point to the attack code

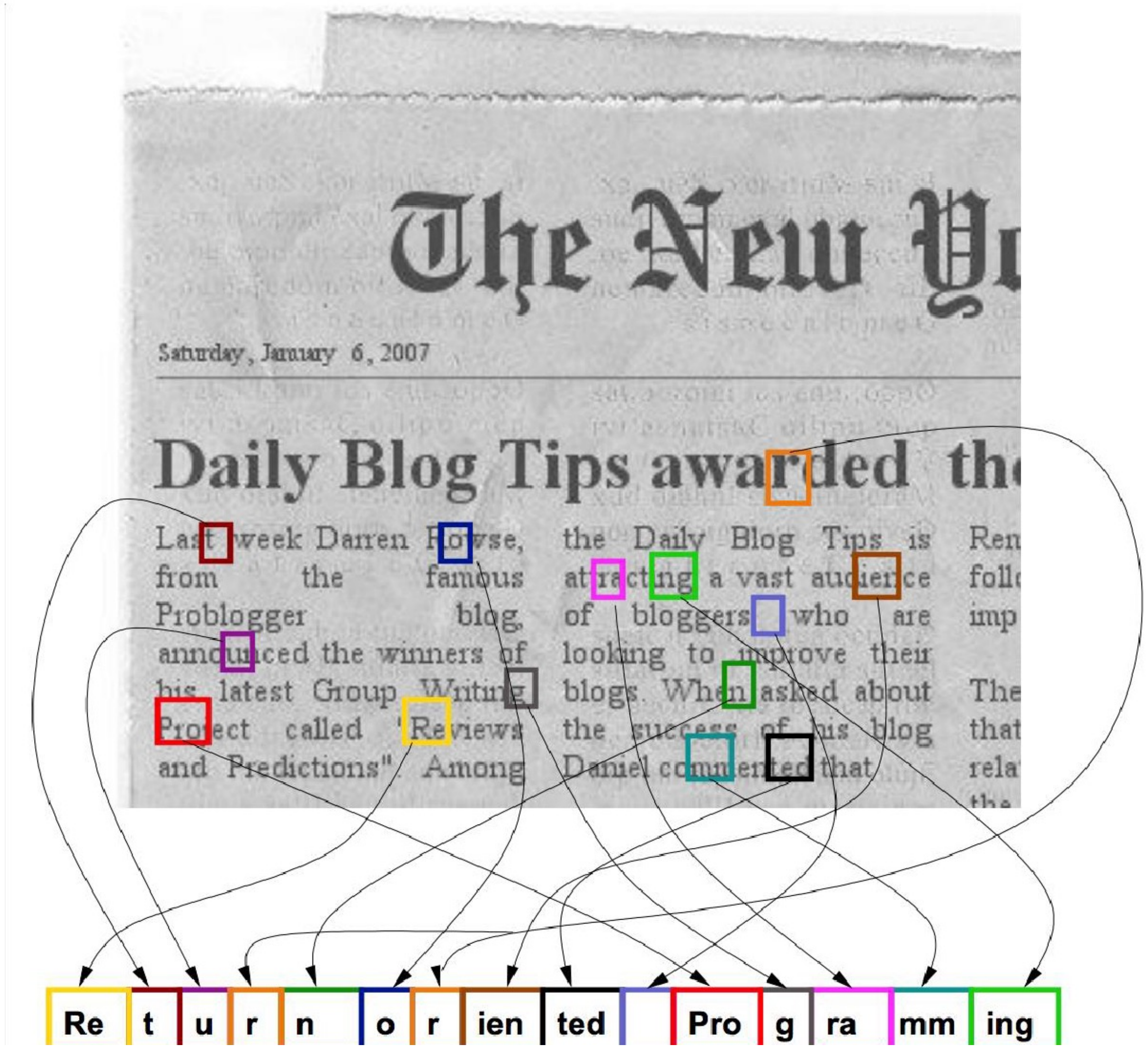
(2) Any memory, on or off the stack, can be modified by a statement that stores a compromised value into the compromised pointer. strcpy(buf, str); *ptr = buf[0];

Other Control Hijacking Opportunities: Frame Pointer



Change the caller's saved frame pointer to point to attacker-controlled memory.
Caller's return address will be read from this memory.

Return-Oriented Programming



Attacks on Non-executable pages

Return into libc: set up the stack and “return” to exec()

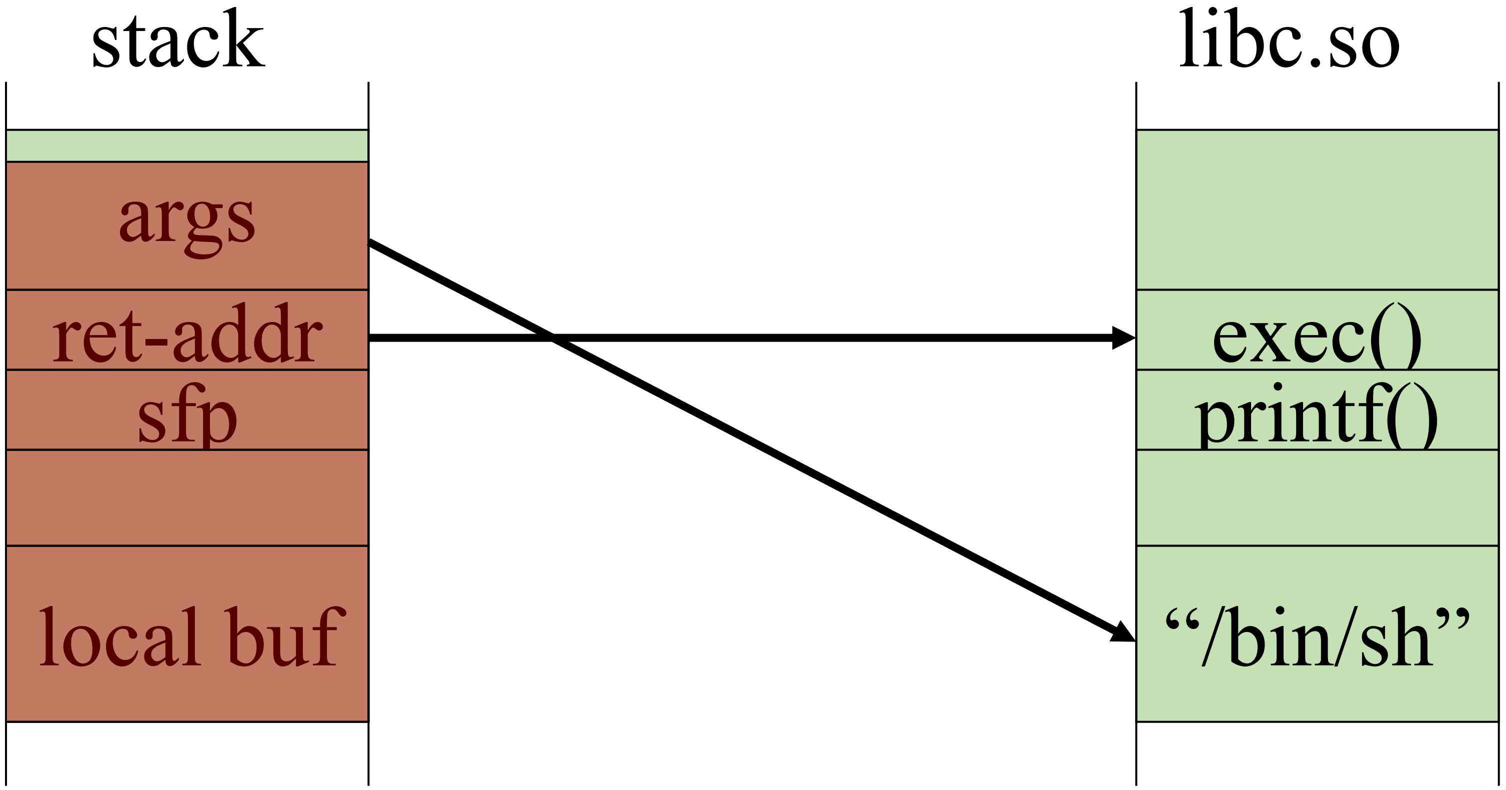
- Overwrite stuff above saved return address with a “fake call stack”, overwrite saved return address to point to the beginning of exec() function
- Especially easy on x86 since arguments are passed on the stack

Return Oriented Programming

- Idea: chain together “return-to-libc” idea many times
- ROP compiler
- Tools democratize things for attackers:
  - Find a set of short code fragments (gadgets) that when called in sequence execute the desired function
  - Inject into memory a sequence of saved "return addresses" that will invoke them Sample gadget: add one to EAX, then return
  - Find enough gadgets scattered around existing code that they’re Turing-complete Compile your malicious payload to a sequence of these gadgets
- *Yesterday's Ph.D. thesis or academic paper is today's Intelligence Agency tool and tomorrow's Script Kiddie download*

Attack: Return Oriented Programming (ROP)

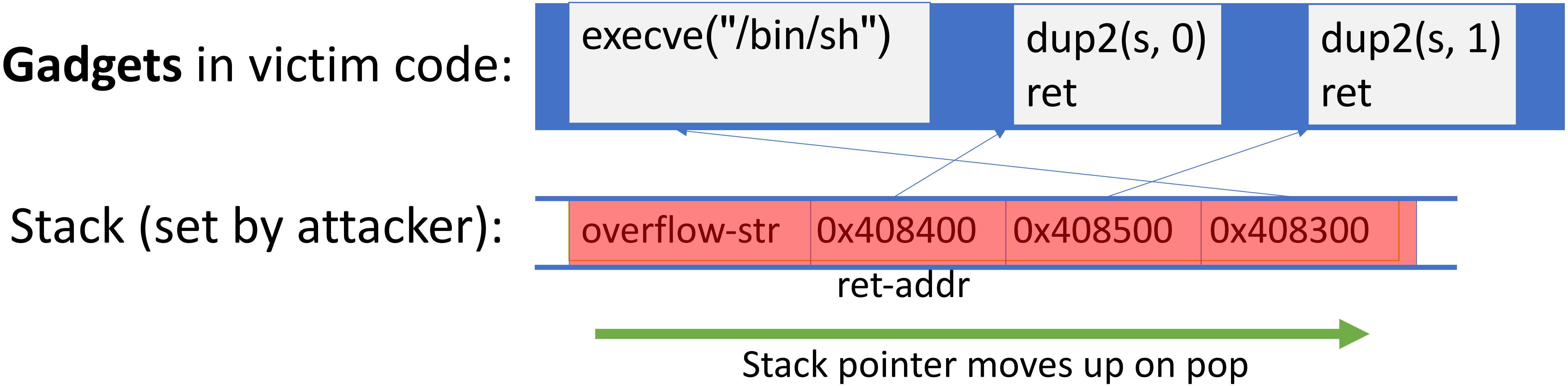
Control hijacking **without injecting code**:



ROP: in more detail

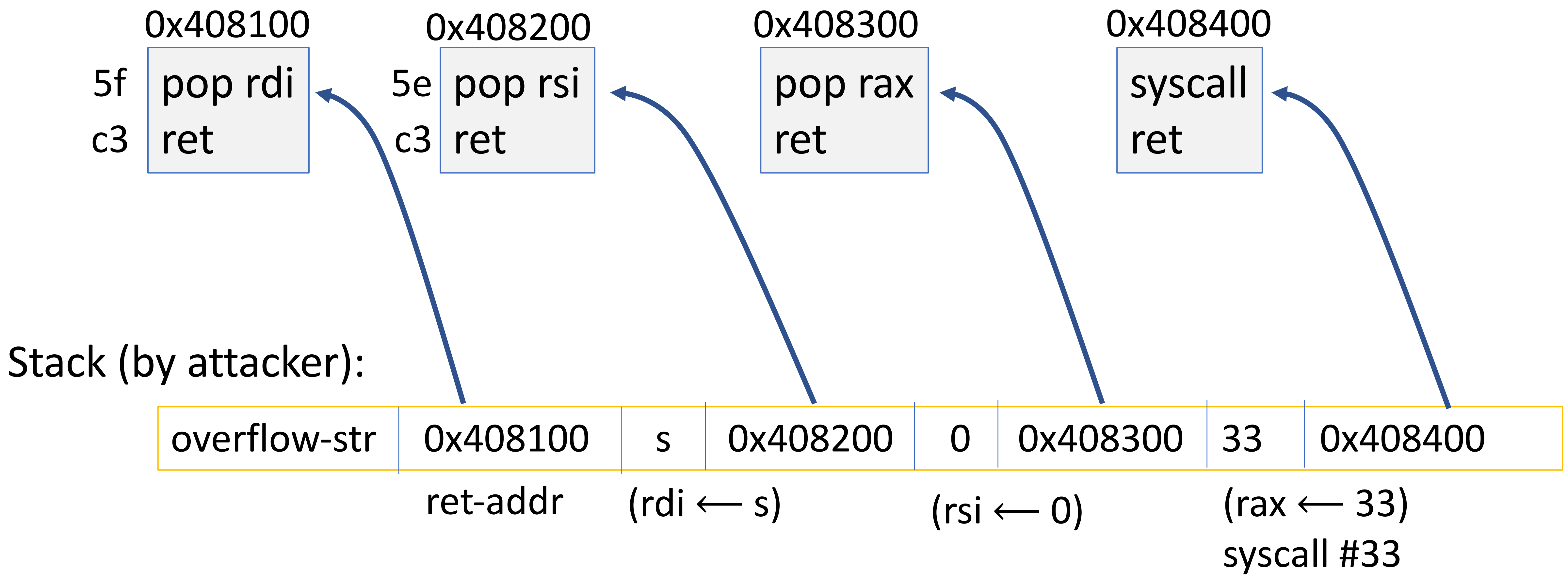
To run /bin/sh we must direct ***stdin*** and ***stdout*** to the socket:

```
dup2(s, 0)      // map stdin to socket
dup2(s, 1)      // map stdout to socket
execve("/bin/sh",  0,  0);
```



ROP: in even more detail

dup2(s,0) implemented as a sequence of gadgets in victim code:



How we safeguard against vulnerabilities as a software engineer?

A. Make buffers (slightly) longer than necessary
B. Safe string manipulation functions (other checks we can do?)
C. Don’t write in C. It’s the root of all evil!
D. As a software programmer there’s only so much we can do… there’s no fix.

Validating input

- Determine acceptable input, check for match --- don’t just check against list of “non-matches”
- Limit maximum length
- Watch out for special characters, escape chars.
- Check bounds on integer values
- Check for negative inputs
- Check for large inputs that might cause overflow!

Validating input

- Filenames
- Disallow *, .., etc.
- Command-line arguments
- Even argv[0]…
- Commands
  - E.g., URLs, http variables., SQL
  - E.g., cross site scripting, (next lecture)

Buffer Overflow: Cures

Idea: prevent execution of untrusted code

- Make stack and other data areas non-executable
  - Note: messes up useful functionality (e.g., Flash, JavaScript)
- Digitally sign all code
- Ensure that all control transfers are into a trusted, approved code image