CS 88: Security and Privacy

06: Software Security: Attacks and Defenses

09-15-2022

SWARTHMORE COLLEGE

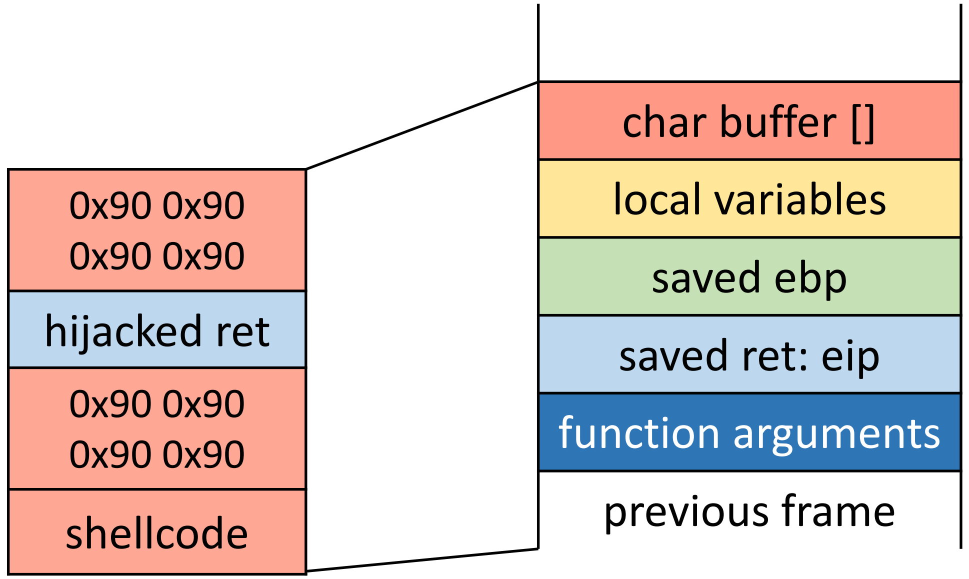
Announcements

Reading Quiz

Draw out a stack diagram and build your very own shellcode attack

Information you are given:

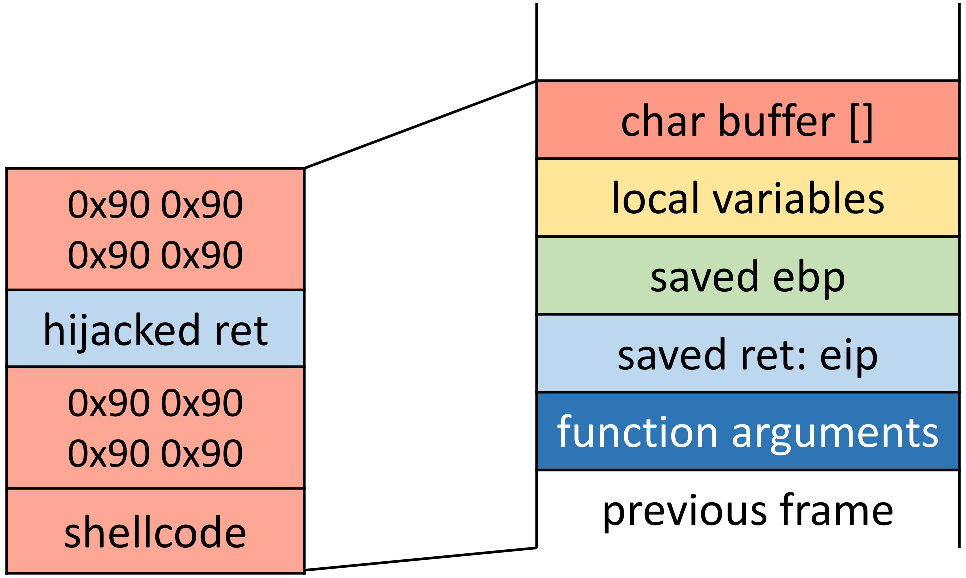
- buffer to overflow:
  - char buffer[50]
  - &buffer[0] = 0xffffd88c
- $eip = 0xffffd8cc
- shellcode = 20 bytes

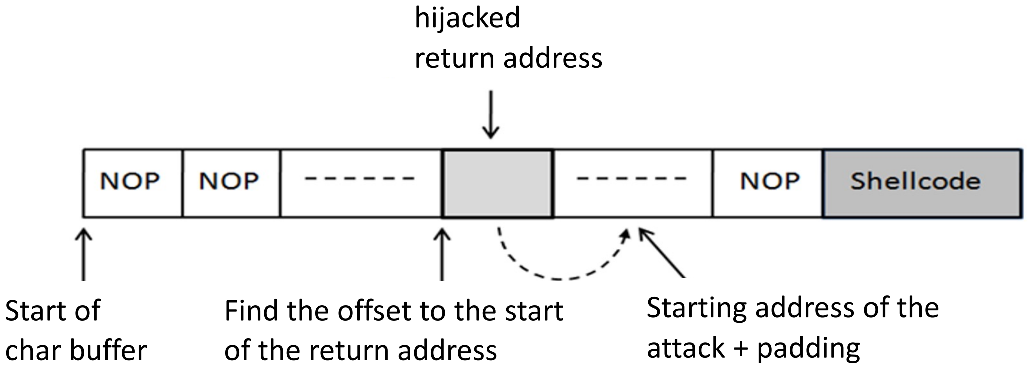


Draw out a stack diagram and build your very own shellcode attack

Information you are given:

- buffer to overflow:
  - char buffer[100]
  - &buffer[0] = 0xffffd88c
- $eip = 0xffffd8bc
- shellcode = 20 bytes





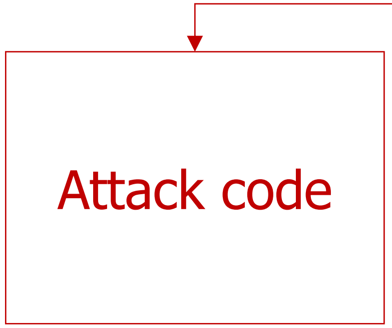
Buffer Overflow: Causes

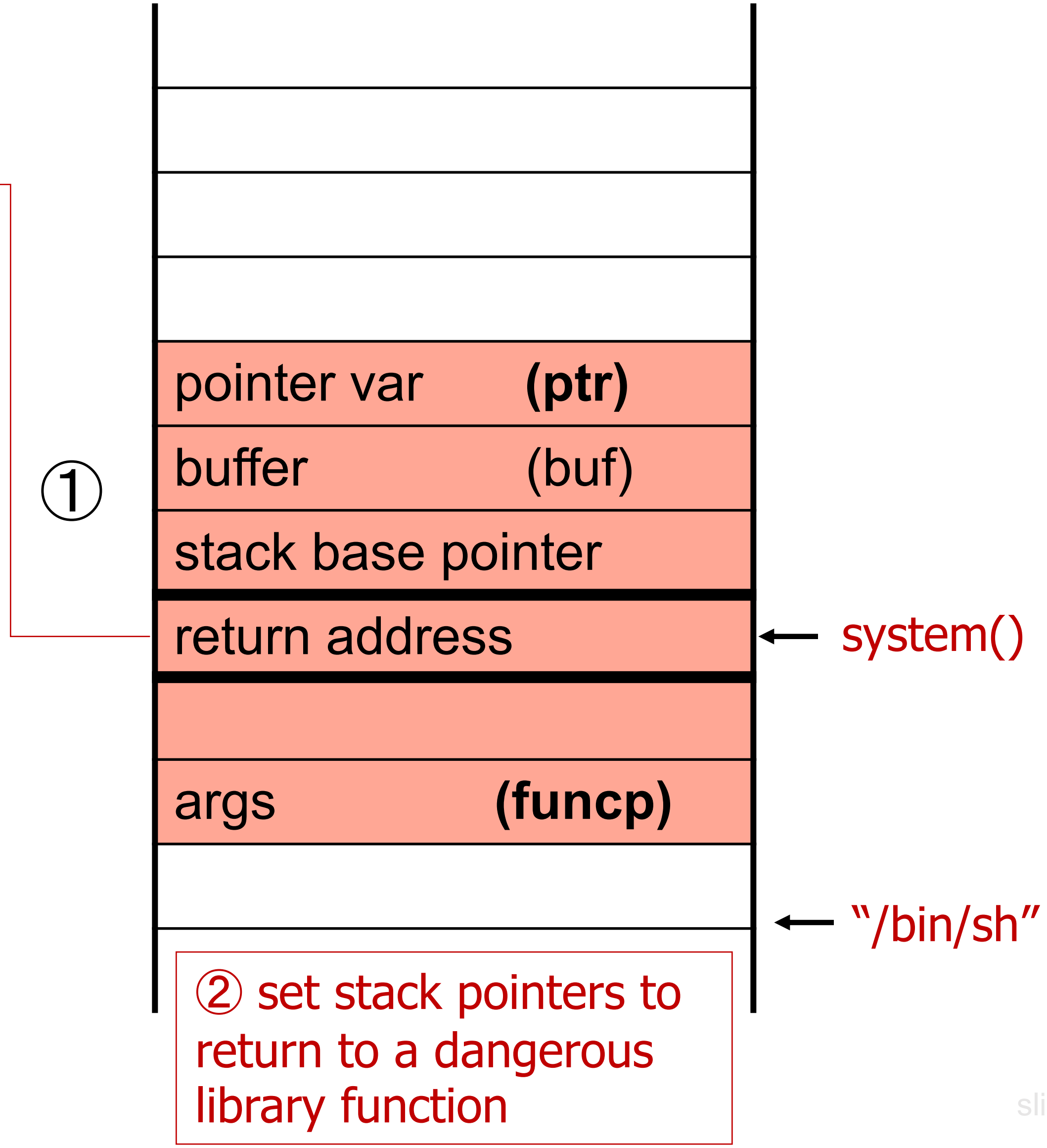
- Typical memory exploit involves code injection
  - Put malicious code at a predictable location in memory, usually masquerading as data
- Trick vulnerable program into passing control to it
  - Overwrite saved EIP, function callback pointer, etc.

Buffer Overflows: can exploit…

A. pointer assignment & memory allocation, de-allocation
B. function pointers
C. calls to library routines
D. general purpose registers
E. format strings

Other Control Hijacking Opportunities: return-to-libc attack

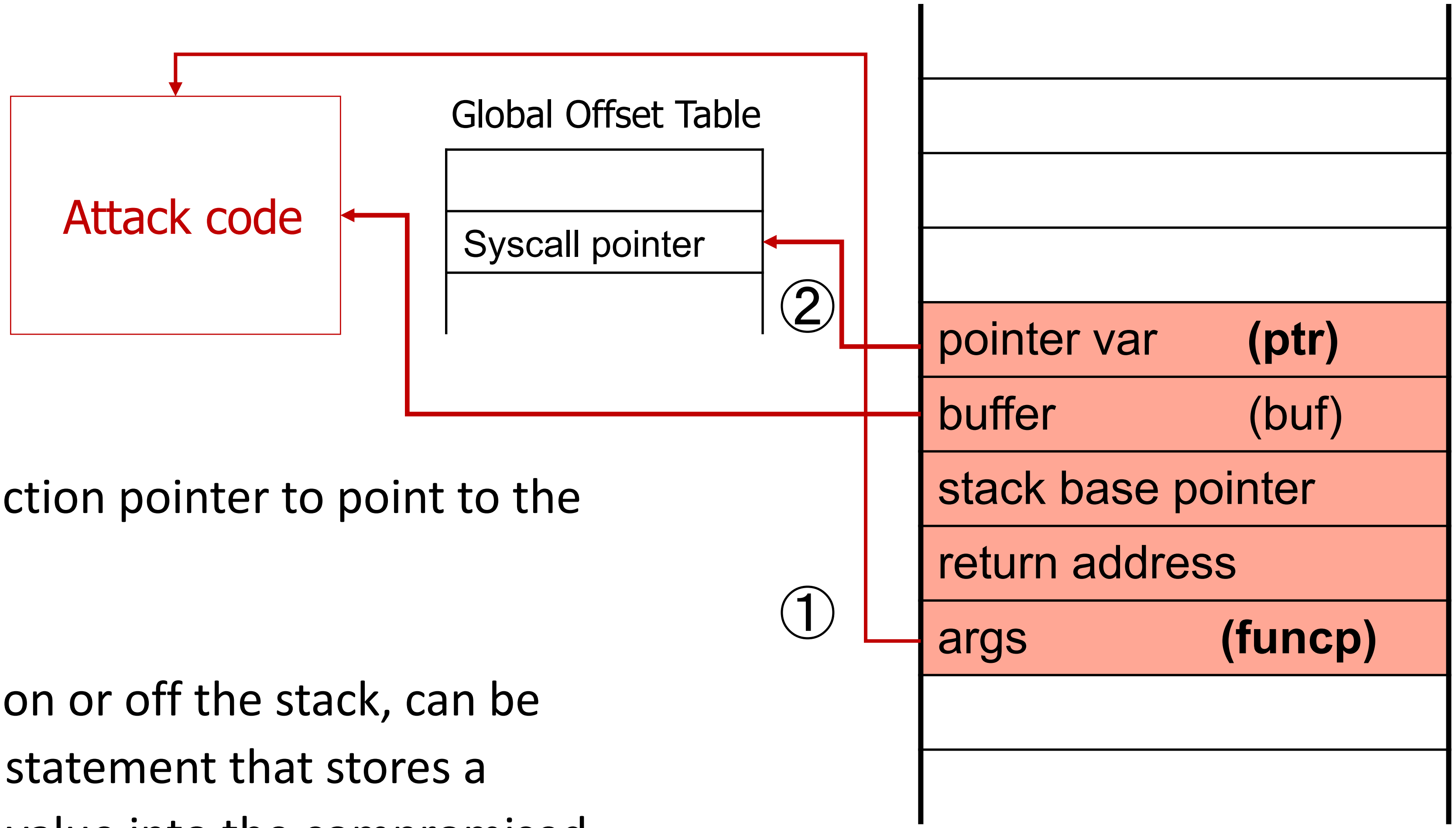




(1) Change the return address to point to the attack code. After the function returns, control is transferred to the attack code.

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

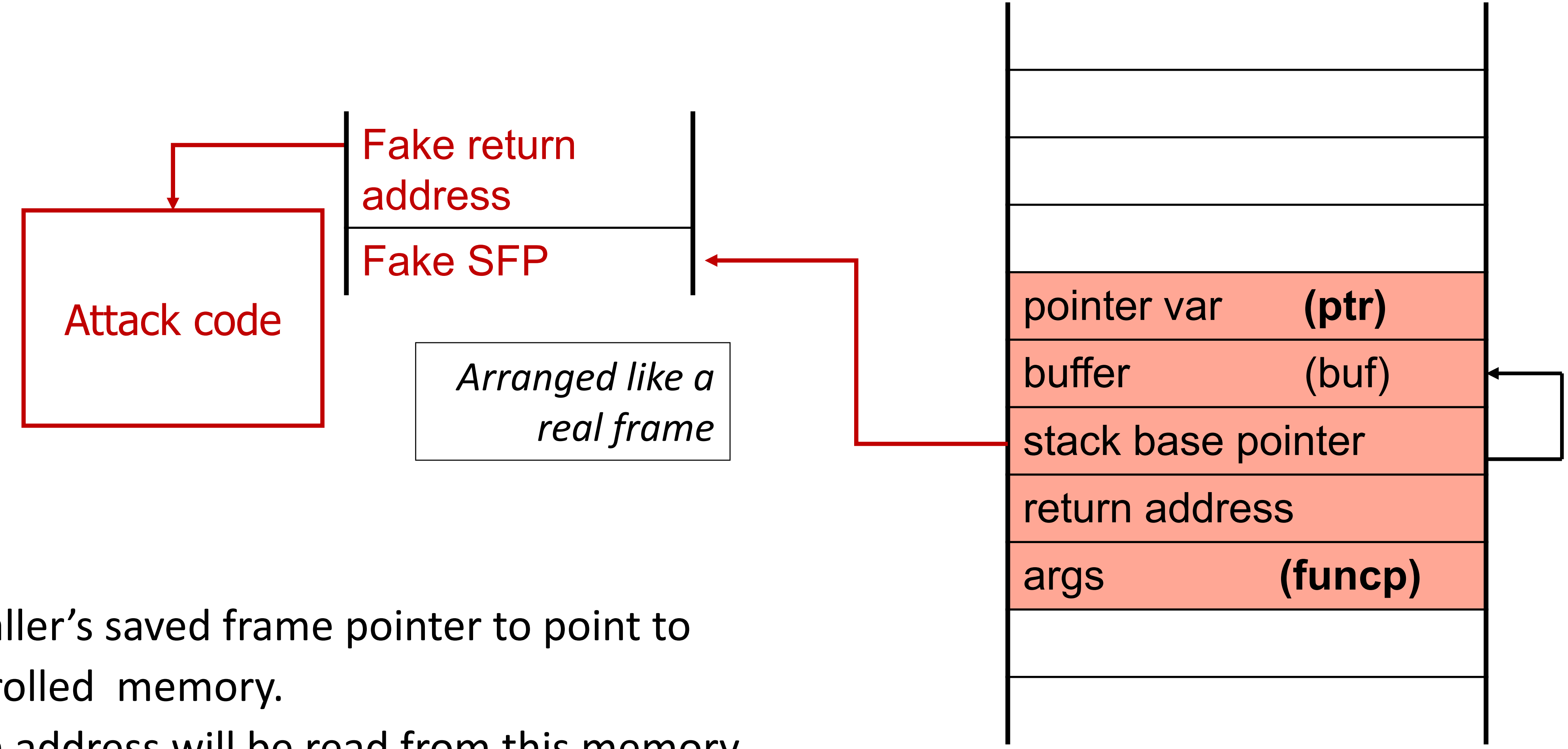
Other Control Hijacking Opportunities: Function Pointers



(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.

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`

Does Range Checking Help?

```
strncpy(char *dest, const char *src, size_t n)
```

- copy no more than n characters from source to destination
- contingent on? the right value of n!

- 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); …
```

A. The fix ensures that there are no vulnerabilities
B. The vulnerabilities are still present.

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;
}
```

A) This code is free from integer overflow vulnerabilities.
B) Integer vulnerabilities still exist.

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;
```

Heartbleed vulnerability

Figure 1. Attack activity related to the Heartbleed vulnerability, as noted for IBM Managed Security Services customers, in April 2014

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!

What damage can an off by 1 error really do?

A) no damage
B) change the value of ebp
C) execute shellcode
D) something else (be prepared to discuss)

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.

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);
}
```

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

A. Nothing
B. Buffer overflow
C. Integer overflow
D. Race Condition

Other overflow targets

- Format strings in C
- Heap management structures used by malloc

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,….)
```

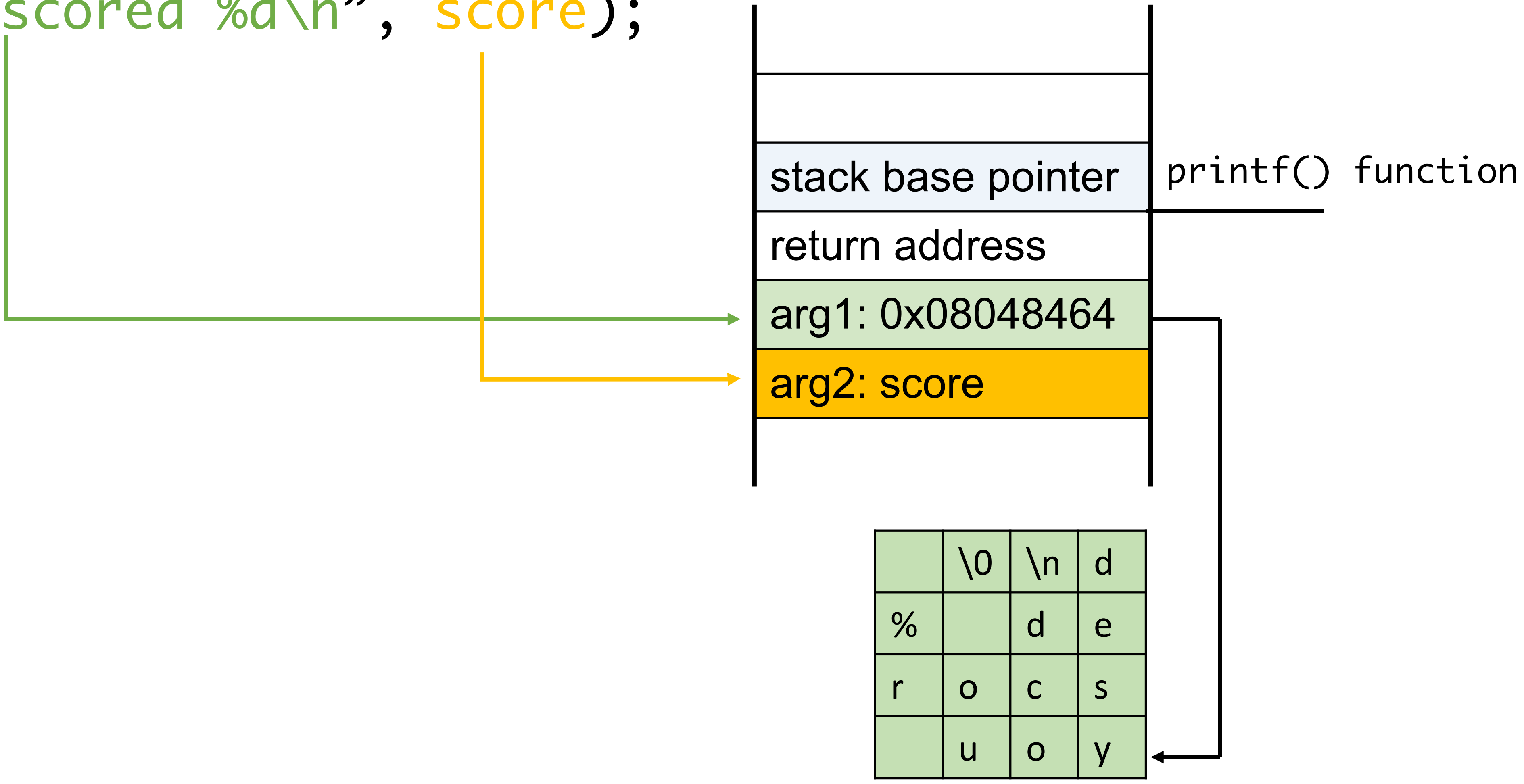
Usage:

```
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);
```



Implementation of printf

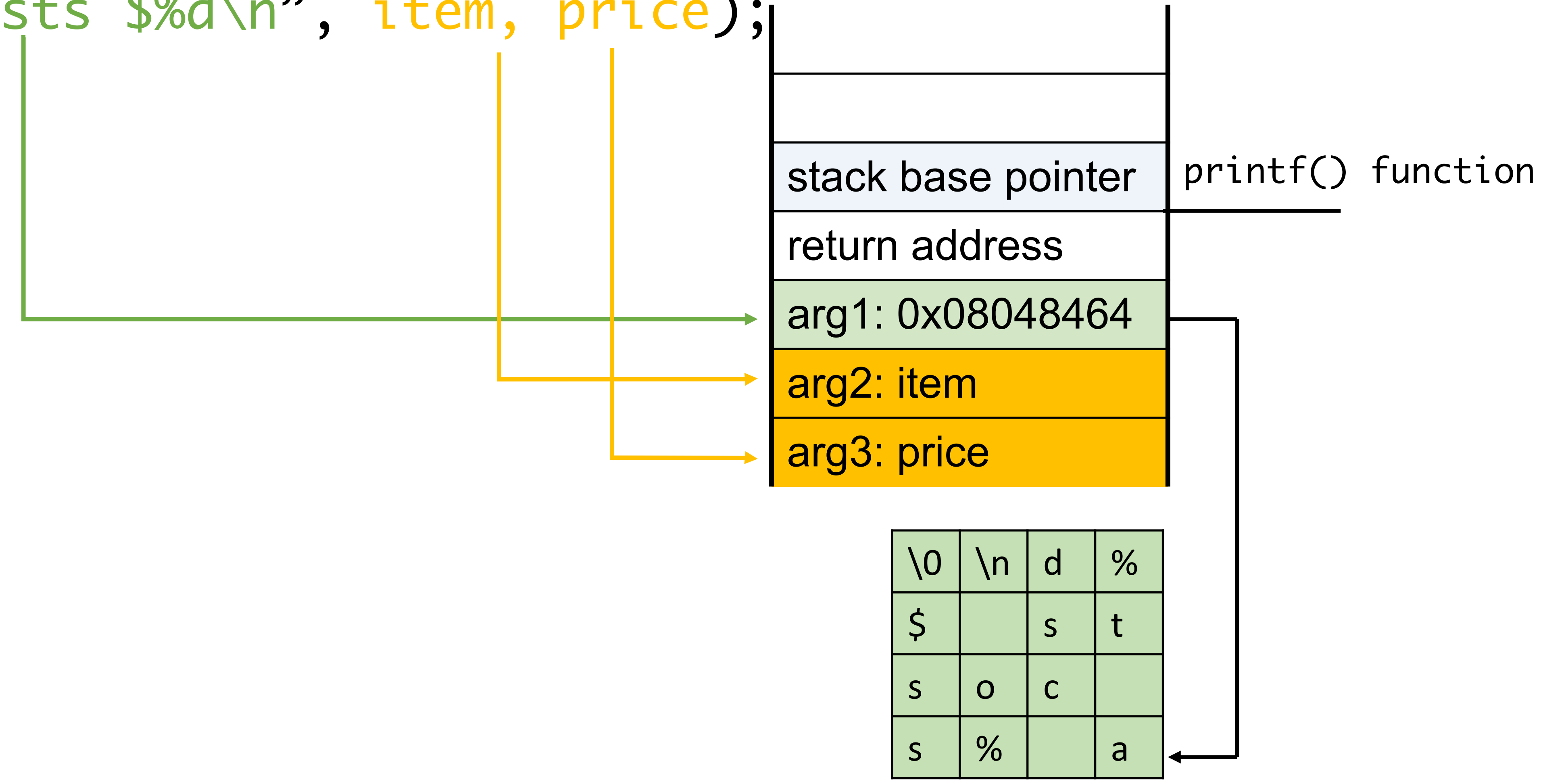
- Special functions va_start, va_arg, va_end compute arguments at run-time

```
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 */
}
```

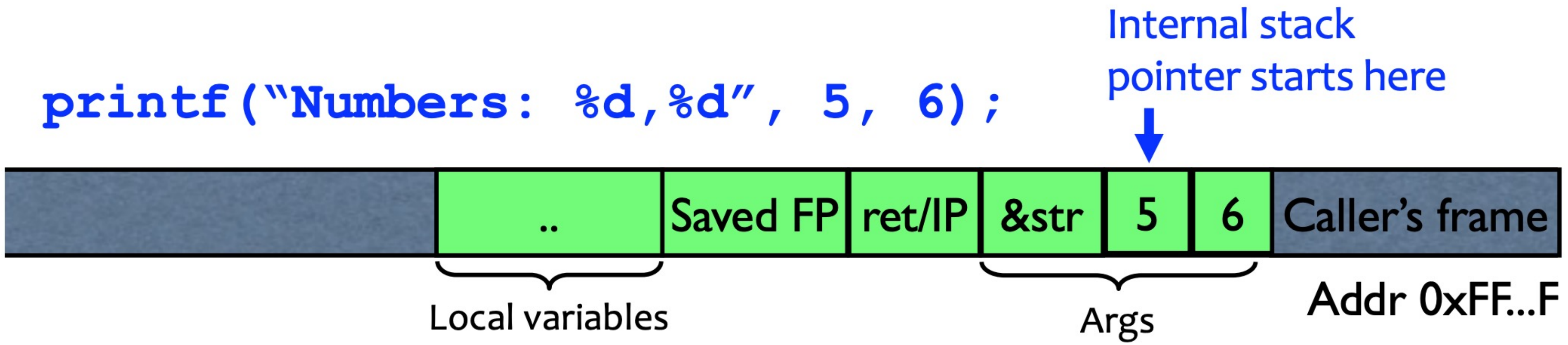
printf has an internal stack pointer

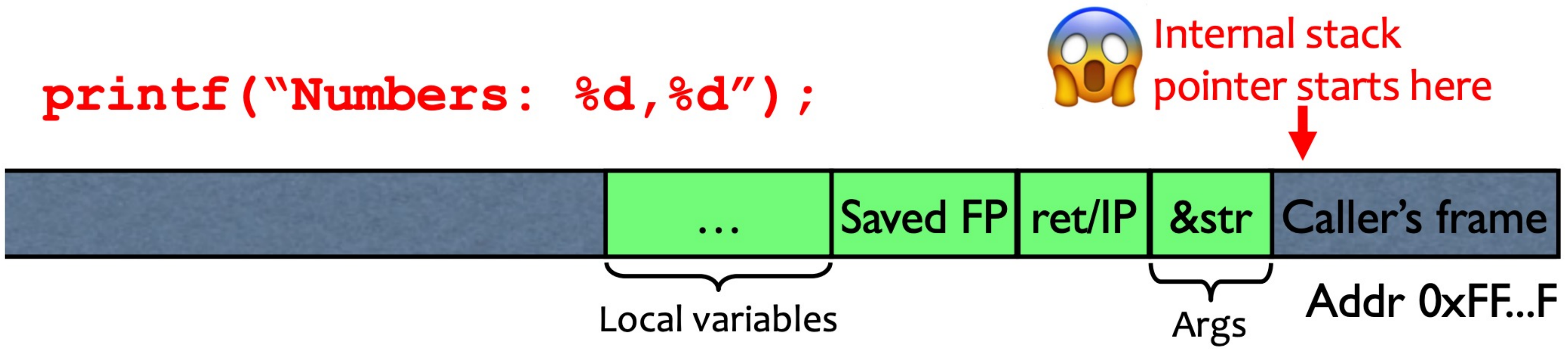
fun with format strings

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



Closer look at the stack

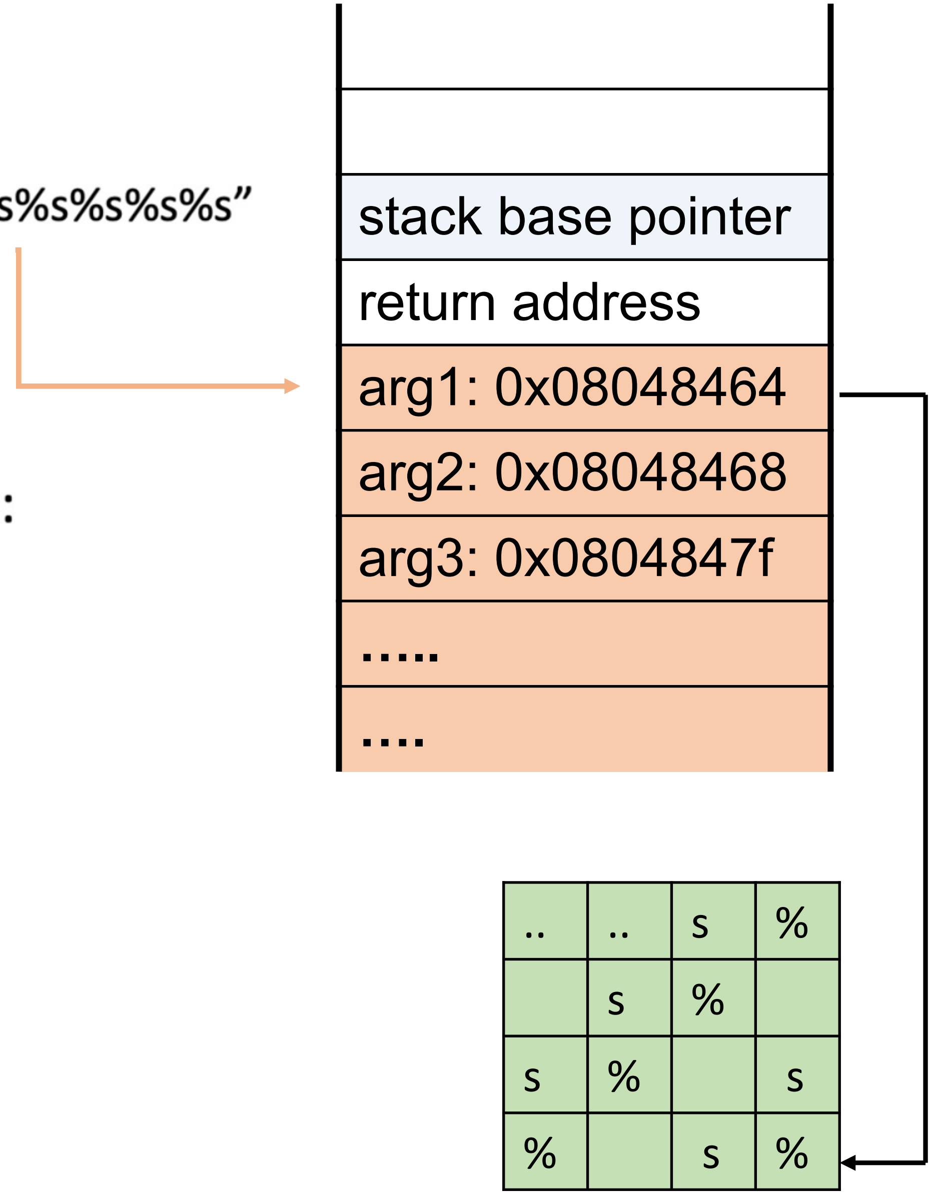




Sloppy use of printf

```
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

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!");
```

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");
```

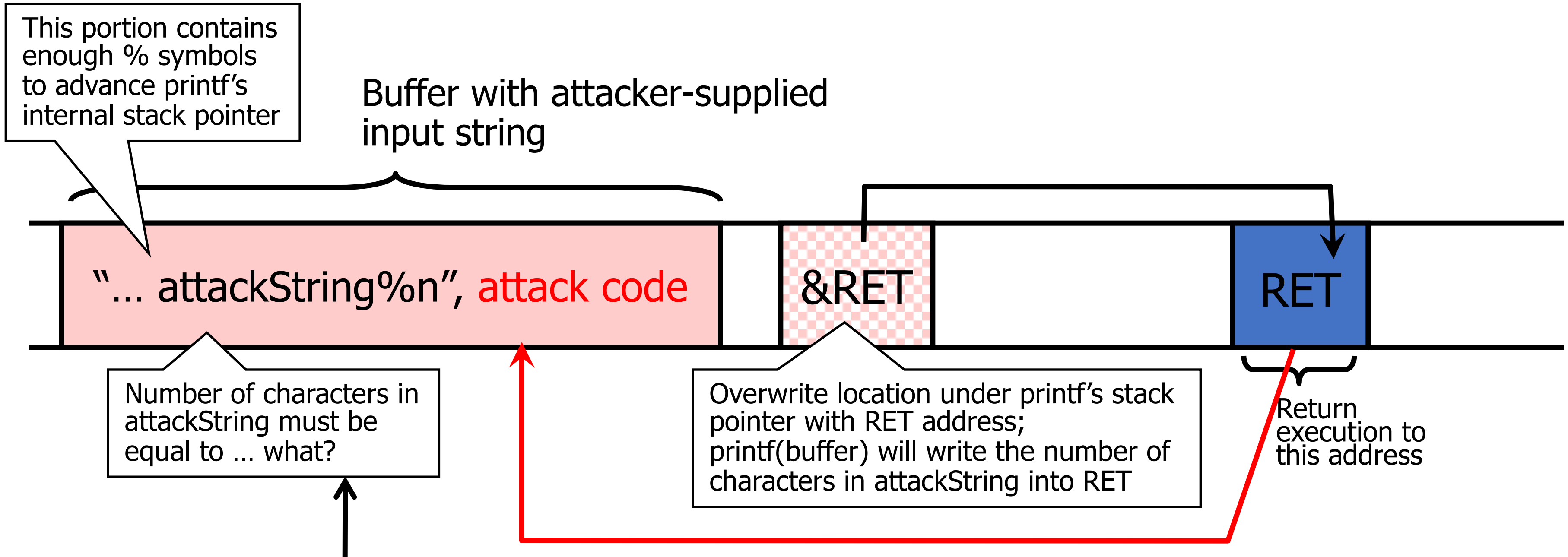
Output

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

instruct printf:

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

Using %n to Mung Return Address



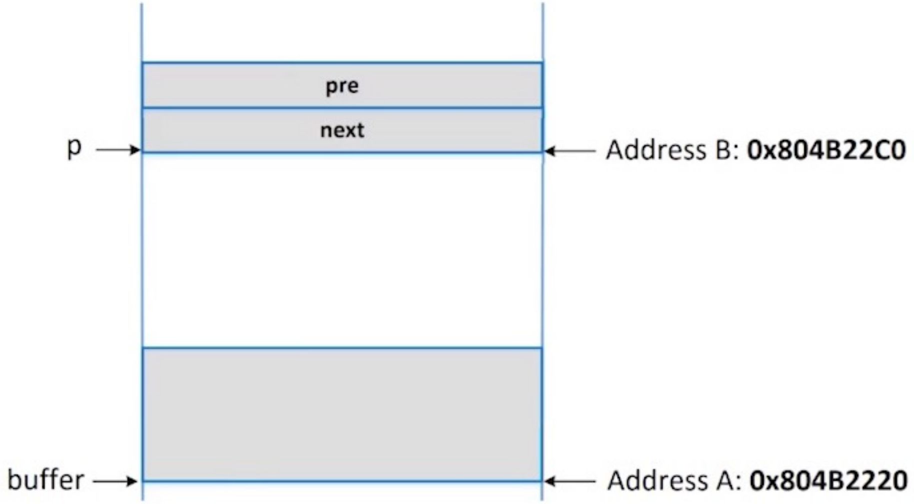
C has a concise way of printing multiple symbols: %Mx will print exactly 4M bytes (taking them from the stack). Attack string should contain enough "%Mx" so that the number of characters printed is equal to the most significant byte of the address of the attack code.

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.

Heap based buffer overflow



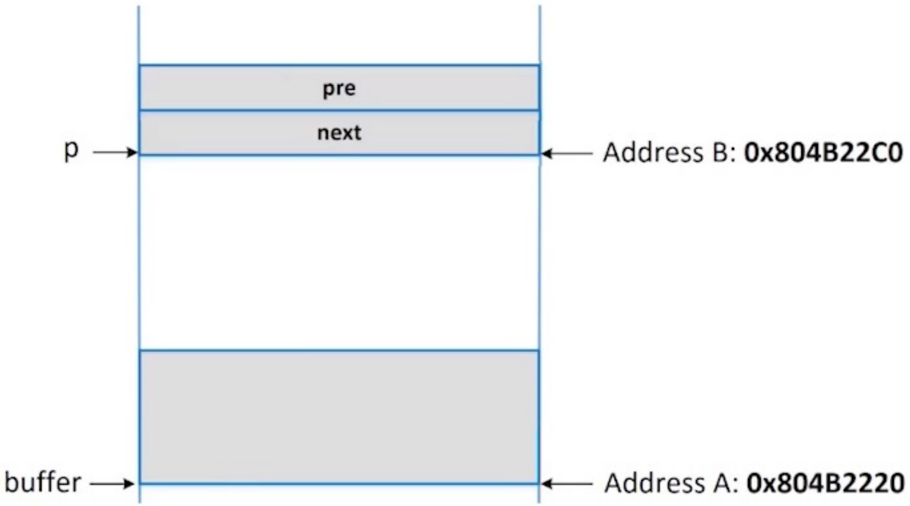

```
struct Node
{
  struct Node *next;
  struct Node *pre;
};
```

```
  // remove Node p from the linked list
  q = p->pre;                ❶
  q->next = p->next;         ❷
```

- Heap stores “chunks” of memory using inked lists
- when malloc is called:
  - stores “meta data” about the chunk right above the newly allocated block
- metadata can be exploited to corrupt memory

Figure by Kevin Du, Syracuse University

Heap Overflow Exploit Techniques




```
struct Node
{
  struct Node *next;
  struct Node *pre;
};
```

```
  // remove Node p from the linked list
  q = p->pre;              ❶
  q->next = p->next;       ❷
```

Overwrite next pointer in linked list
effectively the same as overwriting the return address on the stack
when the malloc function is next involved: control flow is hijacked to point to the attackers code

Heap Buffer Overflow

- a buffer on the heap is not checked
- attacker writes beyond the end of the allocated chunk and corrupts the pointer.

Lots of different variations:

- use after free
- double free
- unlink exploit
- shrinking free chunks..
- house of spirit…

Heaps

Implementation	Platform
ptmalloc2	Linux, HURD (glibc)
SysV AT&T	IRIX, SunOS
Yorktown	AIX
RtlHeap	Windows
tcmalloc	Google and others
jemalloc	FreeBSD, NetBSD, Mozilla
phkmalloc	*BSD

ptmalloc

- Extremely popular malloc (default in glibc)
- Stores memory management metadata inline with user data
  - Stored as small chunks before and after user chunks
- Aggressive optimizations
  - Maintains lists of free chunks binned by size
  - Merges consecutive free chunks to avoid fragmentation

Use after free

Consider the sample code:

```
char *a = malloc(20);     // 0xe4b010
char *b = malloc(20);     // 0xe4b030
char *c = malloc(20);     // 0xe4b050
char *d = malloc(20);     // 0xe4b070

free(a);
free(b);
free(c);
free(d);

a = malloc(20);           // 0xe4b070
b = malloc(20);           // 0xe4b050
c = malloc(20);           // 0xe4b030
d = malloc(20);           // 0xe4b010
```

The state of the particular fastbin progresses as:

1. 'a' freed.
   > head -> a -> tail
2. 'b' freed.
   > head -> b -> a -> tail
3. 'c' freed.
   > head -> c -> b -> a -> tail
4. 'd' freed.
   > head -> d -> c -> b -> a -> tail
5. 'malloc' request.
   > head -> c -> b -> a -> tail ['d' is returned]
6. 'malloc' request.
   > head -> b -> a -> tail ['c' is returned]
7. 'malloc' request.
   > head -> a -> tail ['b' is returned]
8. 'malloc' request.
   > head -> tail ['a' is returned]

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

Double free

Consider this sample code:

```
a = malloc(10);     // 0xa04010
b = malloc(10);     // 0xa04030
c = malloc(10);     // 0xa04050

free(a);
free(b);  // To bypass "double free or corruption (fasttop)
free(a);  // Double Free !!

d = malloc(10);     // 0xa04010
e = malloc(10);     // 0xa04030
f = malloc(10);     // 0xa04010   - Same as 'd' !
```

The state of the particular fastbin progresses as:

1. 'a' freed.

   > head -> a -> tail

2. 'b' freed.

   > head -> b -> a -> tail

3. 'a' freed again.

   > head -> a -> b -> a -> tail

4. 'malloc' request for 'd'.

   > head -> b -> a -> tail ['a' is returned]

5. 'malloc' request for 'e'.

   > head -> a -> tail ['b' is returned]

6. 'malloc' request for 'f'.

   > head -> tail ['a' is returned]

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

How we safeguard against buffer overflows 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)