CS 88: Security and Privacy

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

02-01-2024

SWARTHMORE COLLEGE

Announcements

- Clicker registrations posted – let me know if I don’t have yours
- Please choose partnerships for Lab 1 (EdStem) – last chance.
- Reading quizzes count from this week
- Lab 0 is due today
- Midterm dates on edstem later today

Today

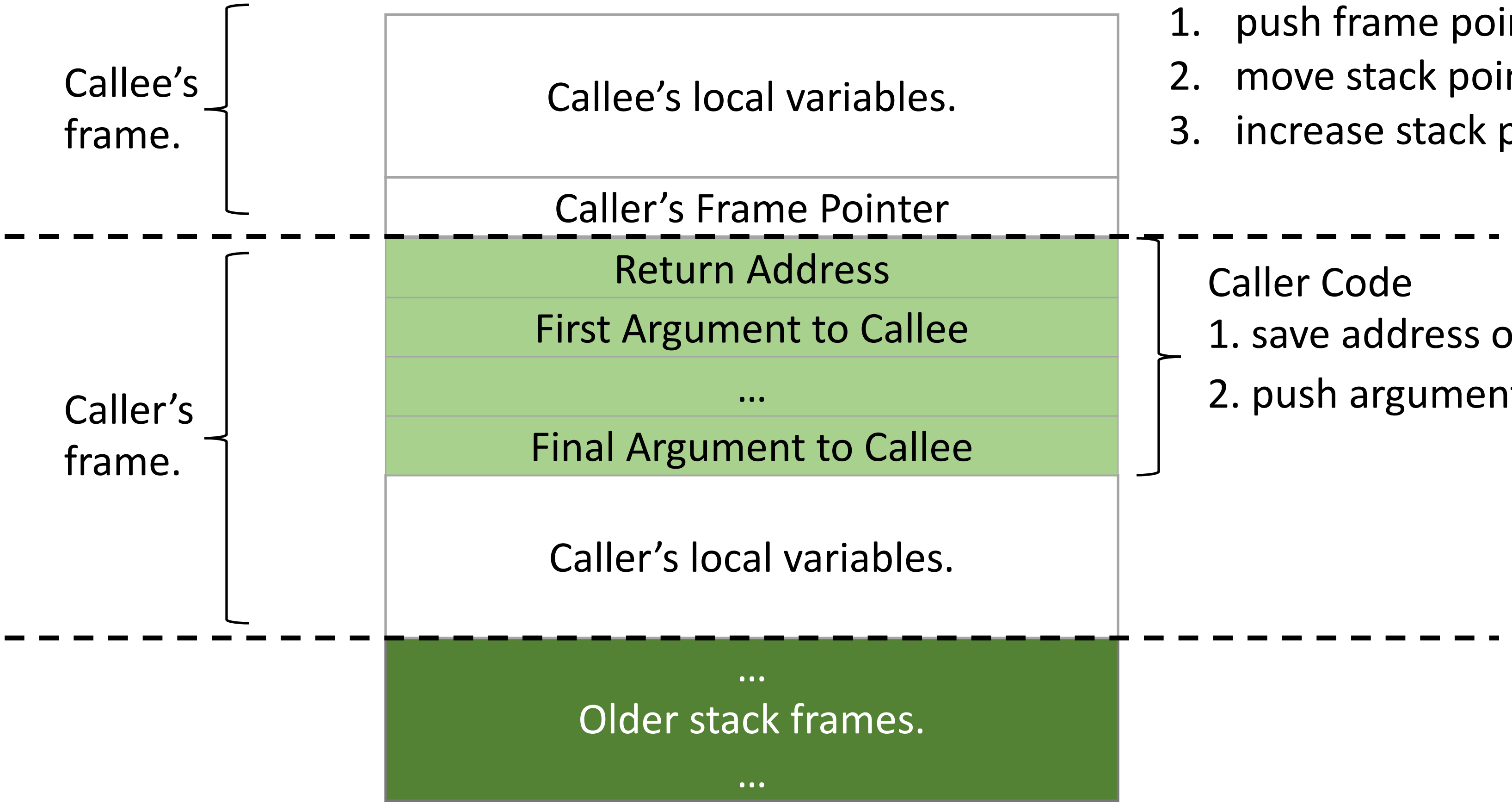
- Software attacks
    - Integer Overflow Attacks
    - Format String Attacks
    - Heap overflow (shelphish)

Buffer Overflows

Putting it all together…

Callee Code

1. push frame pointer
2. move stack pointer to frame pointer
3. increase stack pointer



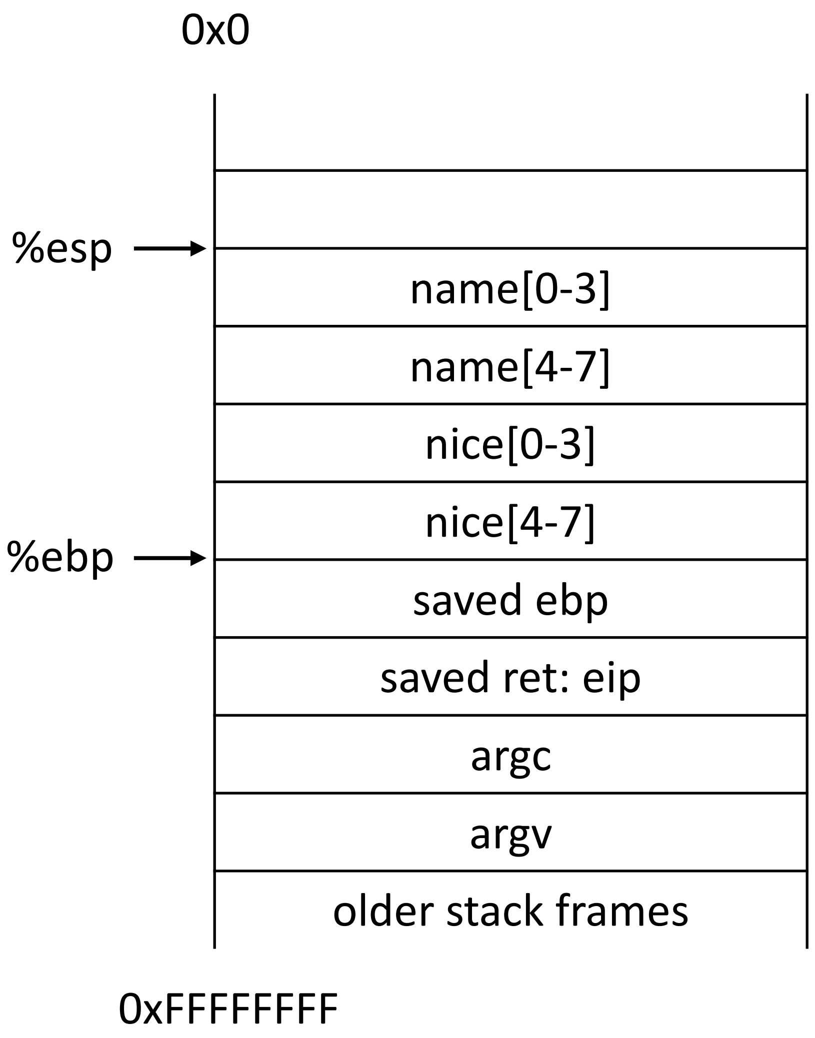
Caller Code

1. save address of next instruction
2. push arguments

Example 1

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

int main(int argc, char**argv){
    char nice[] = “is nice.”;
    char name[8];
    gets(name);
    printf(“%s %s\n”, name, nice);
    return 0;
}
```

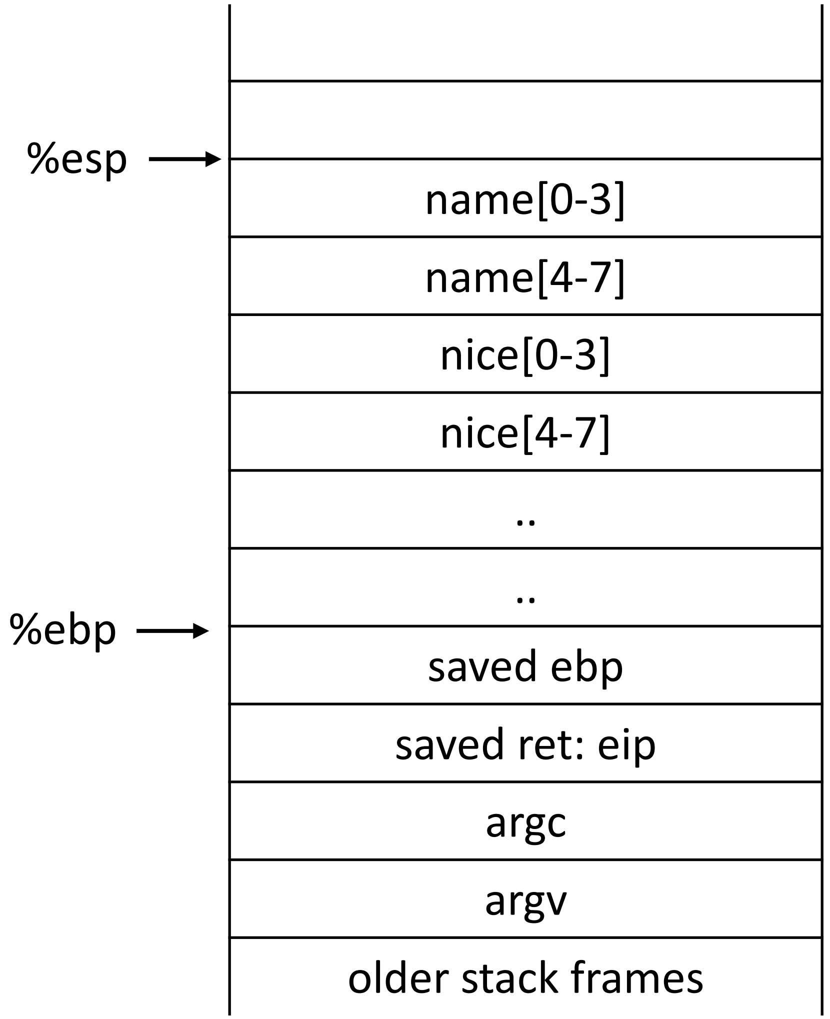


Function call stack

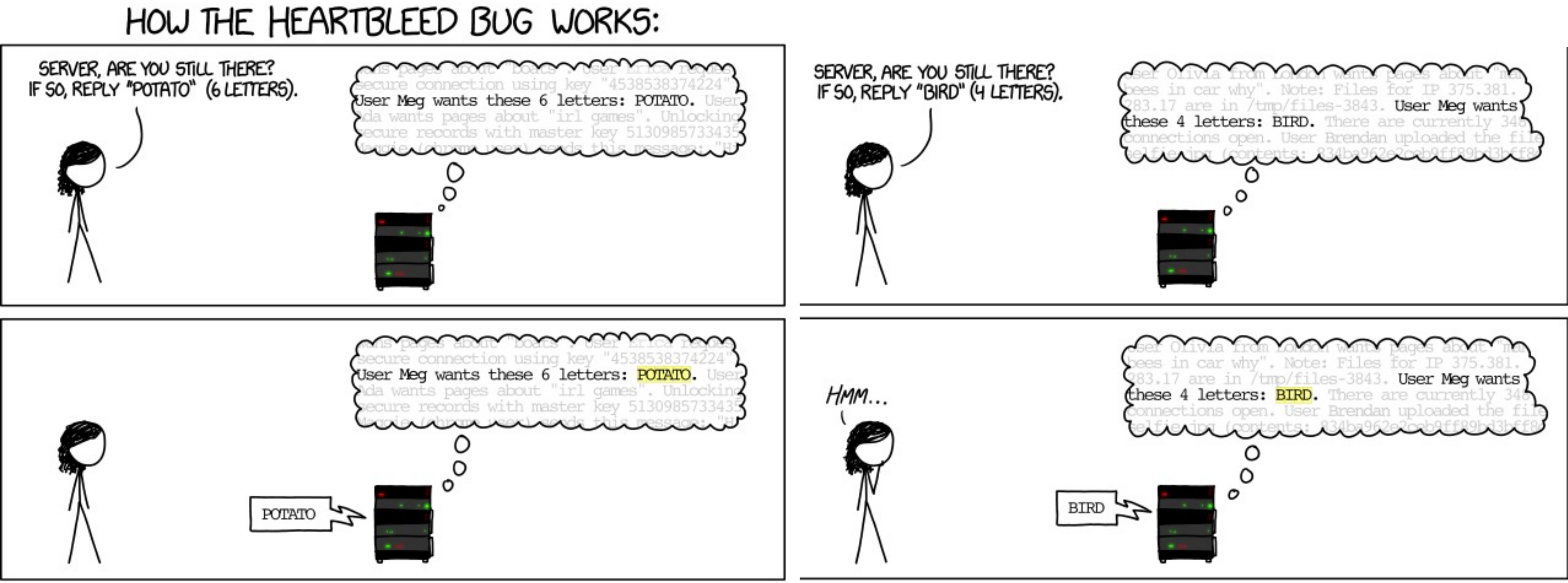
What happens if we read a long name?

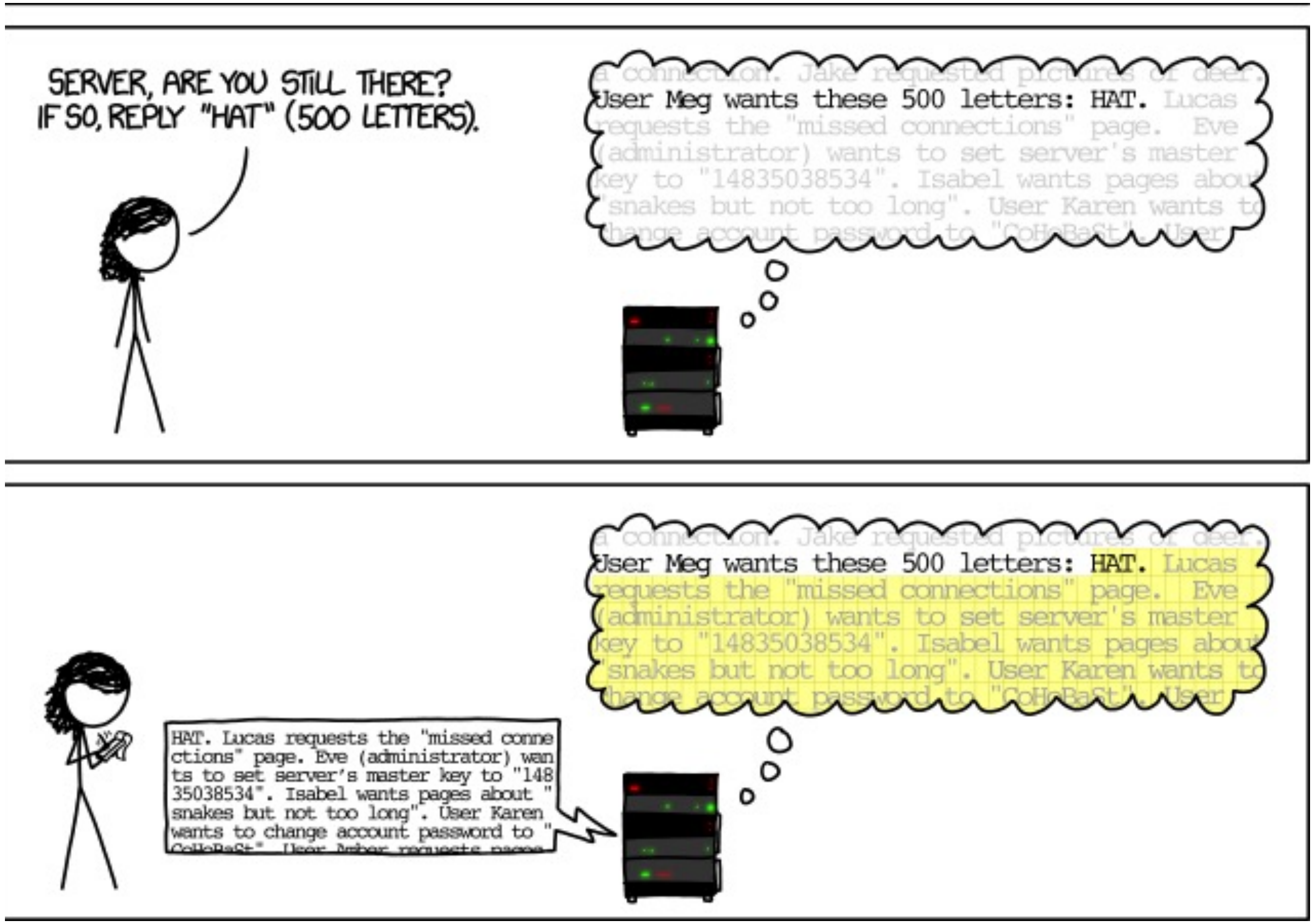
```c
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv){
   char nice[] = “is nice.”;
   char name[8];
   gets(name);
   printf(“%s %s\n”, name, nice);
   return 0;
}
```



A. Nothing bad will happen
B. Something nonsensical will result
C. Something terrible will result


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
ecure connection using key "4538538374224"
User Meg wants these 6 letters: POTATO. User
da wants pages about "irl games". Unlocking
ecure records with master key 5130985733435
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
bees in car why". Note: Files for IP 375.381.
283.17 are in /tmp/files-3843. User Meg wants
these 4 letters: BIRD. There are currently 346
connections open. User Brendan uploaded the file
HMM...
BIRD


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages abou snakes but not too long". User Karen wants to
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages abou snakes but not too long". User Karen wants to
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "

Buffer Overflow example

```
#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;
}
```

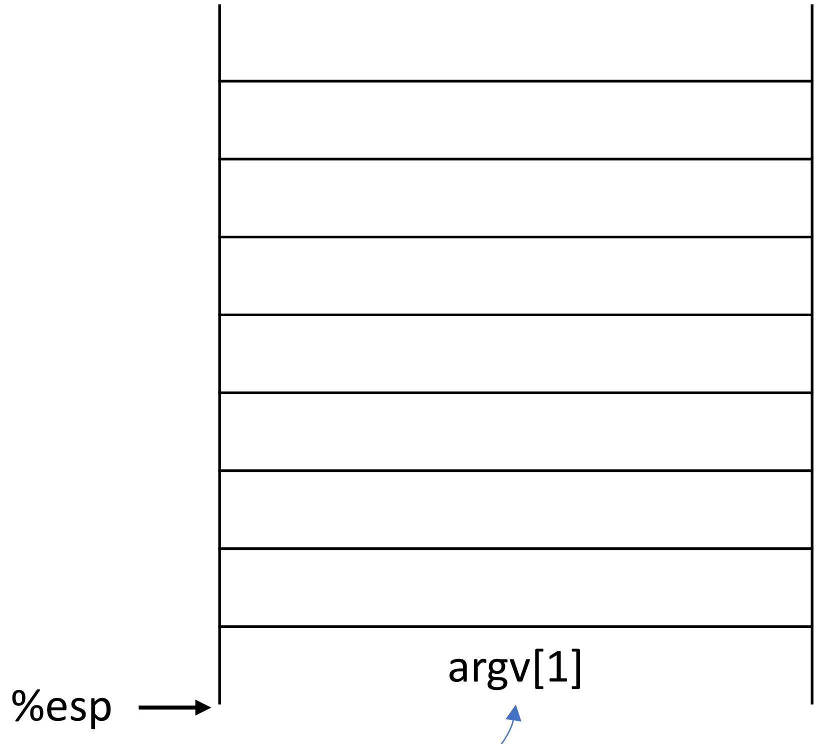
Buffer Overflow example

```c
#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;
}
```



Load function arguments starting with the last argument

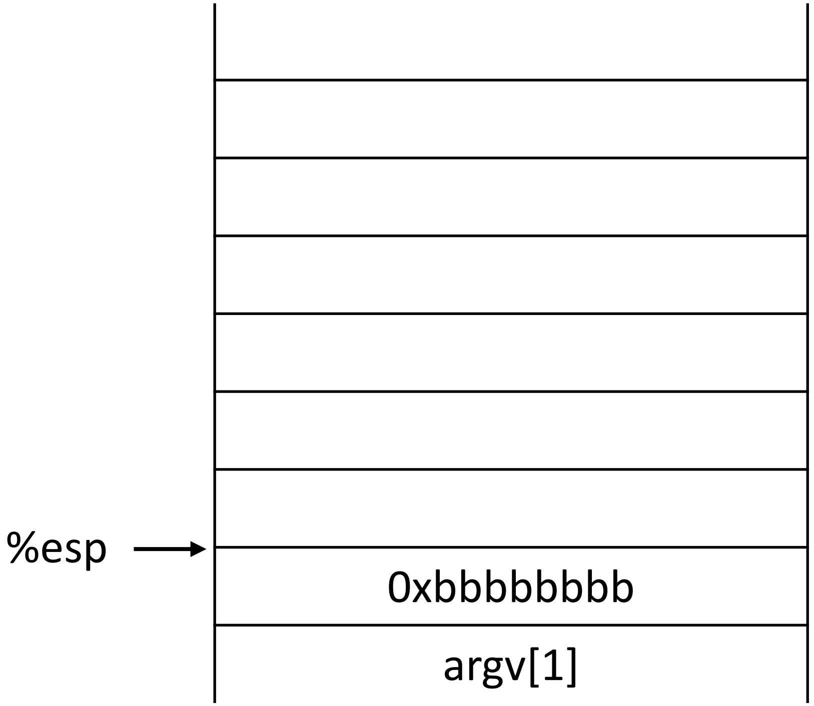
Buffer Overflow example

```
#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;
}
```



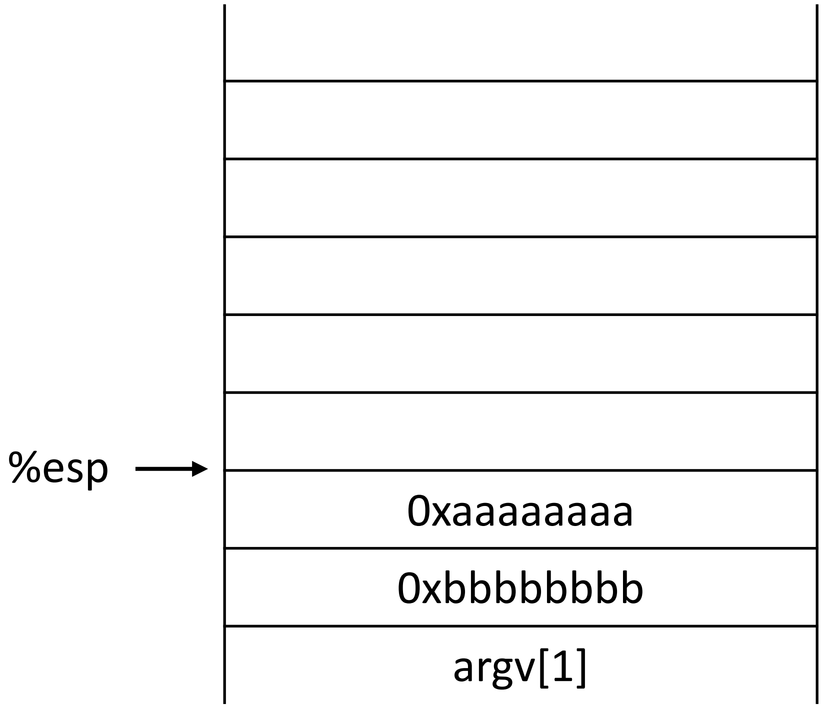
Buffer Overflow example

```
#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;
}
```



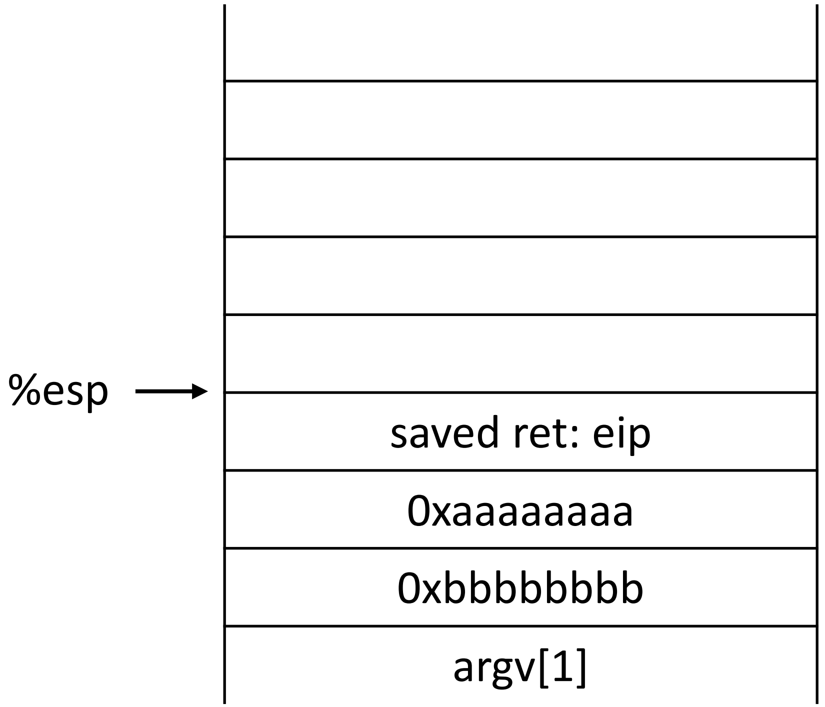
Buffer Overflow example

```
#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;
}
```



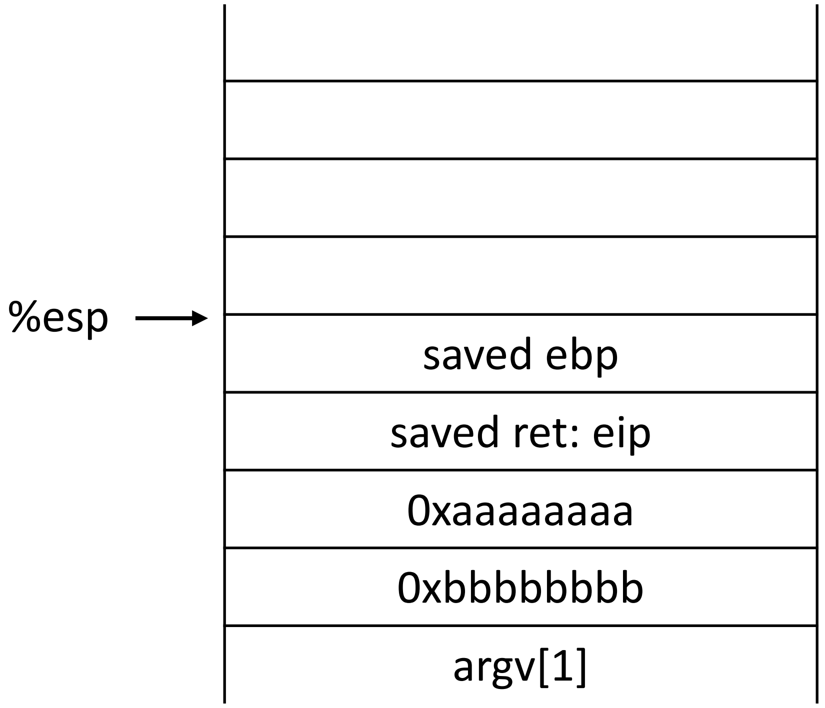
Buffer Overflow example

```c
#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;
}
```



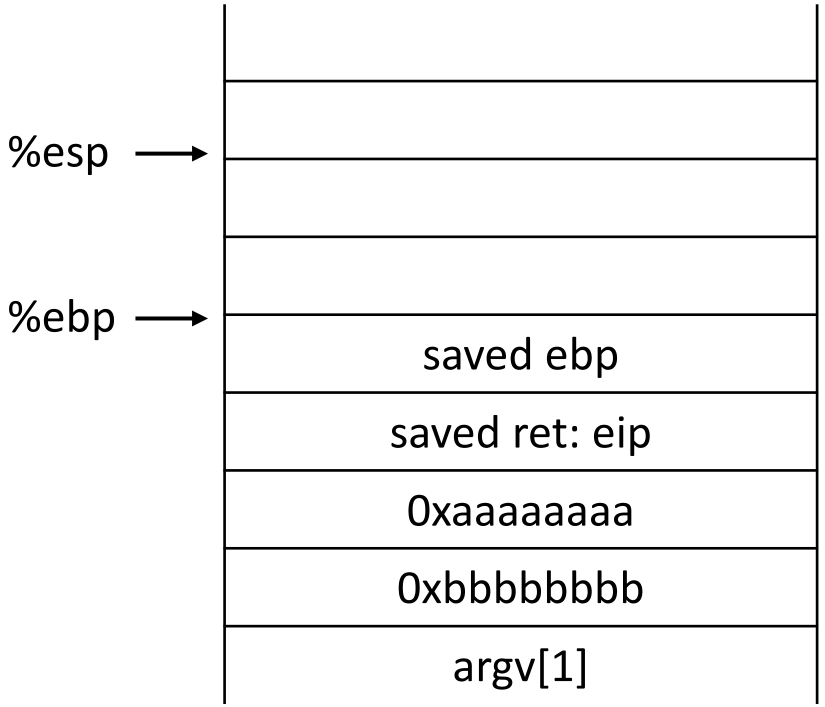
Buffer Overflow example

```
#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;
}
```



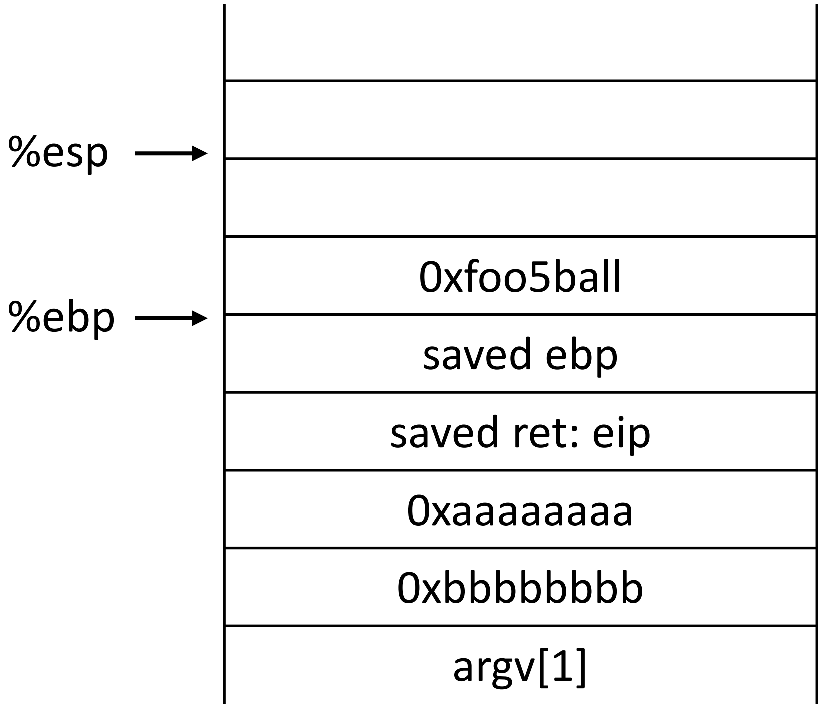
Buffer Overflow example

```c
#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;
}
```



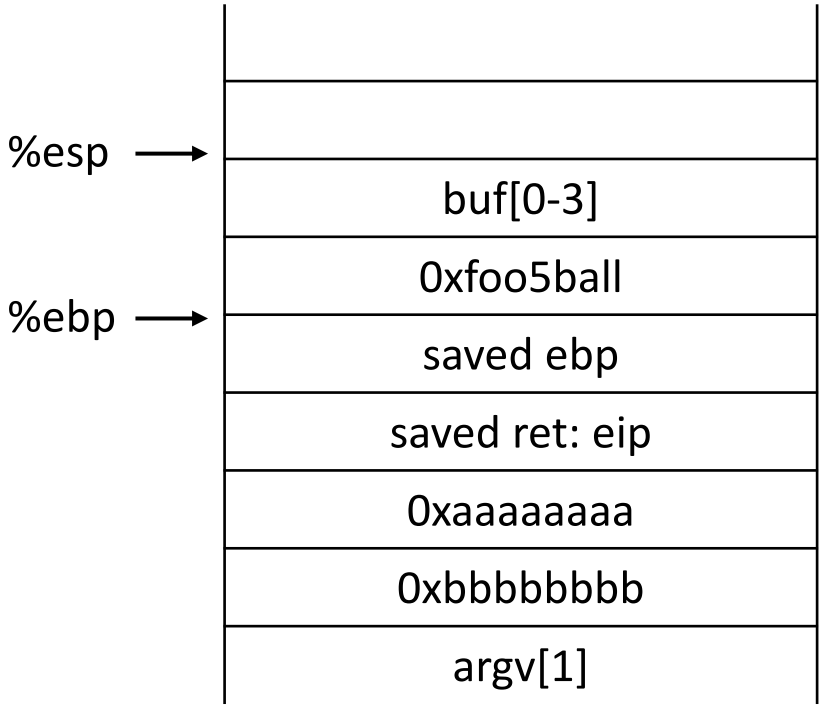
Buffer Overflow example

```
#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;
}
```



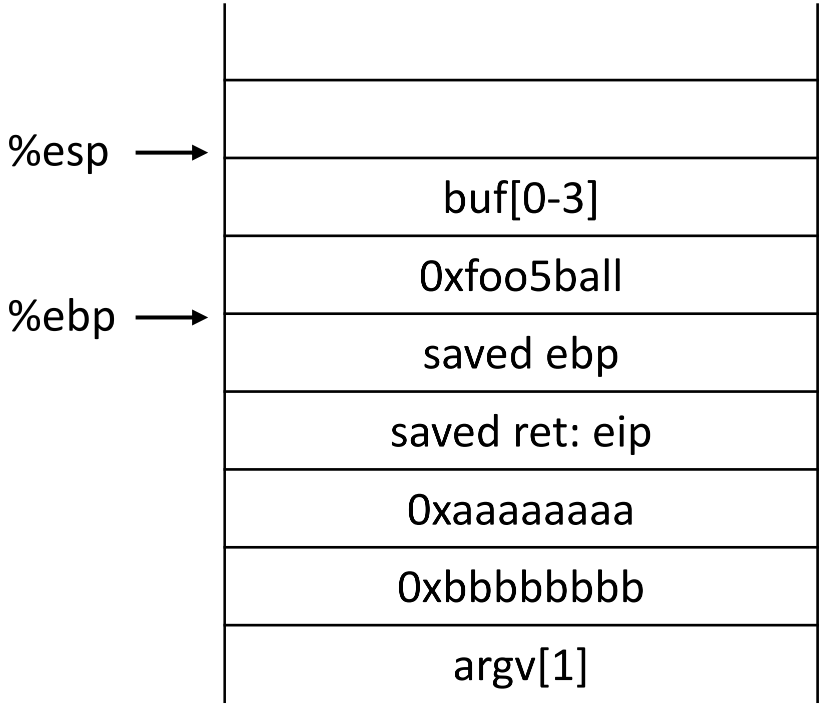
Buffer Overflow example

```
#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;
}
```



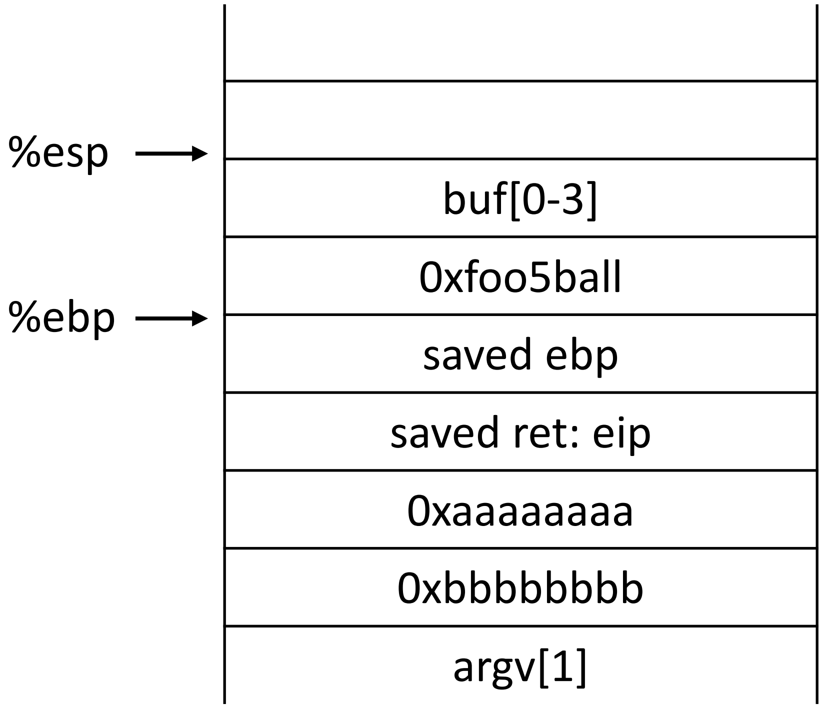
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAA"

```c
#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;
}
```



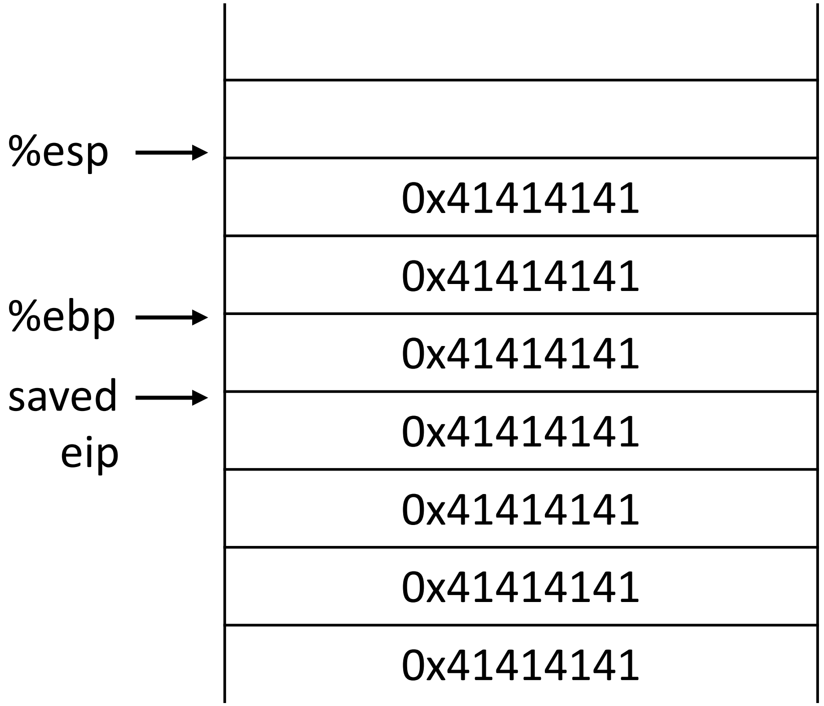
Buffer Overflow example: If the first input is "AAAAAAAAAAAAAAAA"

```c
#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;
}
```



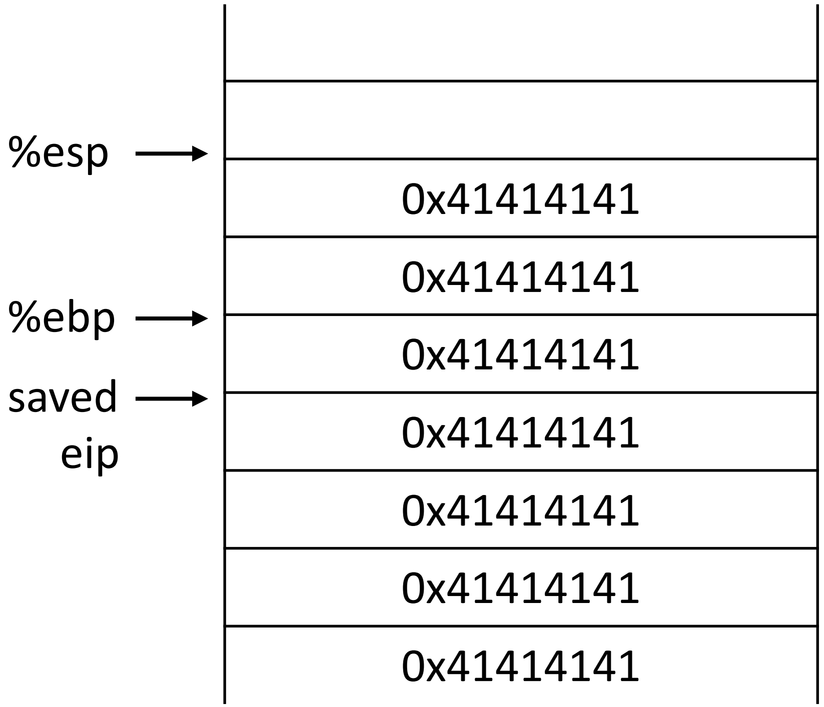
Buffer Overflow example: If the first input is “AAAAAAAAAAAAAAAA”

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

void foo() { 0x08049b95
  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;
}
```



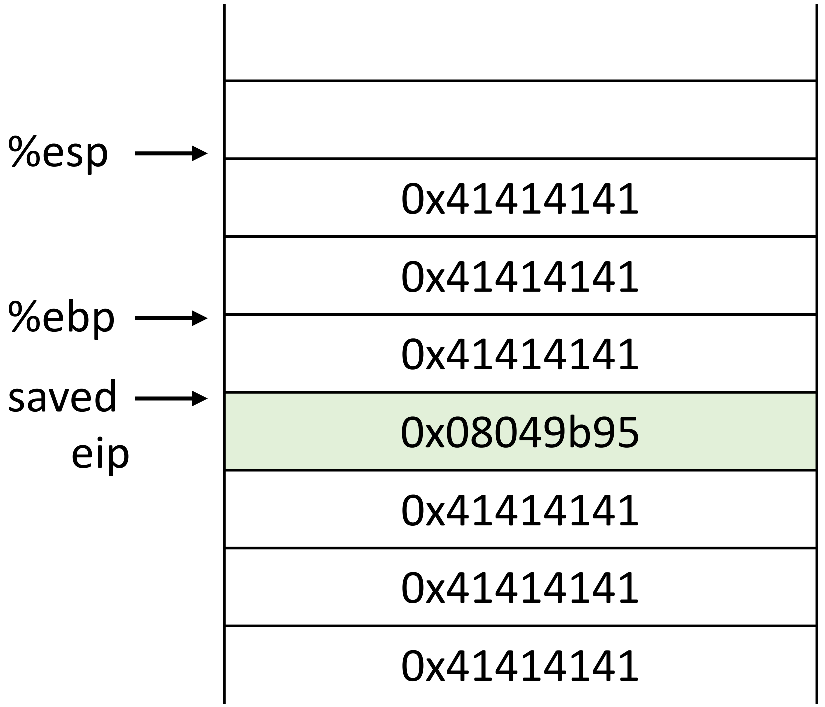
Buffer Overflow example: If the first input is “AAAAAAAA\x95\x9b\x04\x08”

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

void foo() { 0x08049b95
  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;
}
```



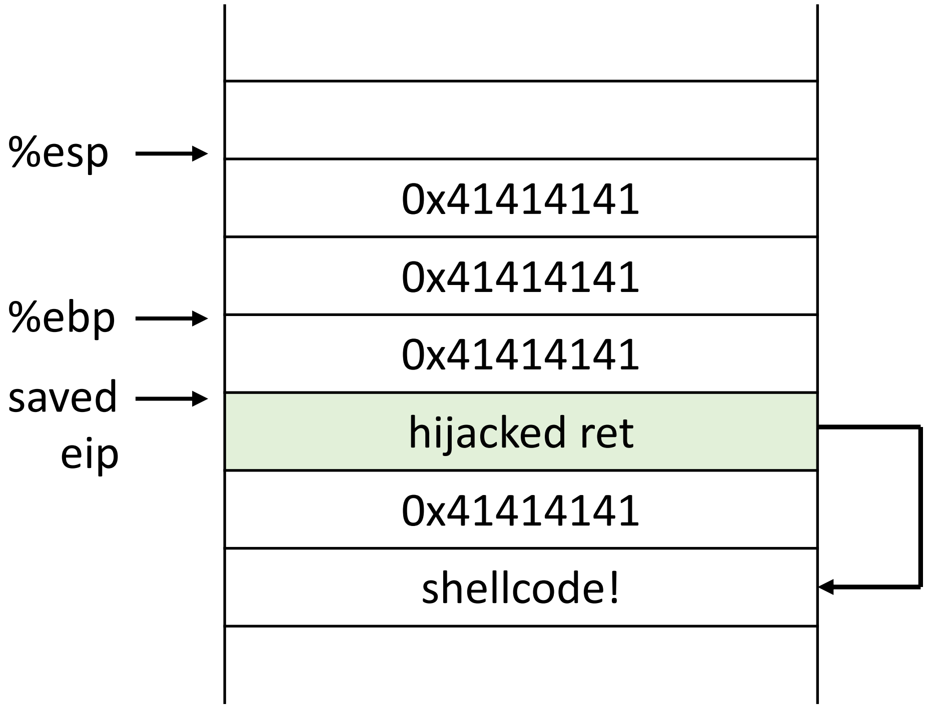
Better Hijacking Control

```
#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;
}
```



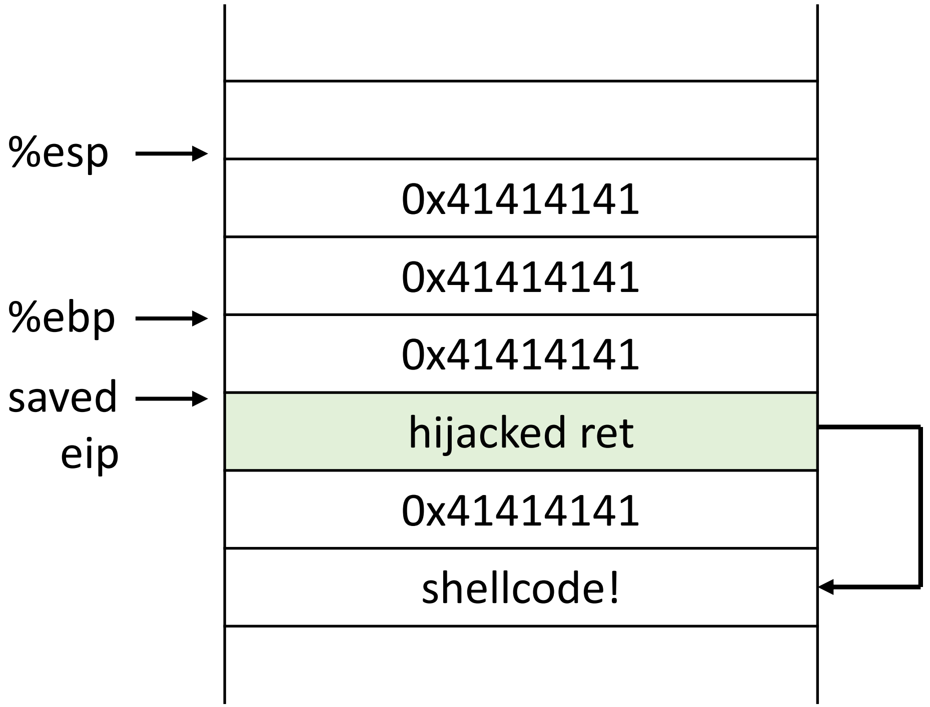
Better Hijacking Control

```
#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

Shellcode

- Type of control flow hijack: taking control of the instruction pointer
- Small code fragment to which we transfer control
- Shellcode used to execute a shell

Shellcode

```
int main(void) {
  char* name[1];
  name[0] = “/bin/sh“;
  name[1] = NULL;
  execve(name[0], name, NULL);
  return 0;
}
```

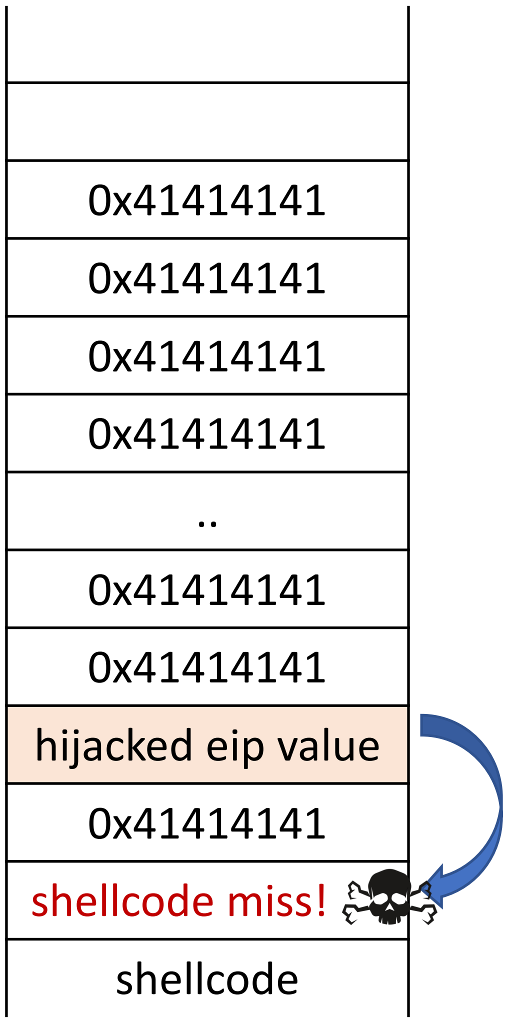
How do we transfer this to code?
Take the compiled assembly?

Payload is not always robust

Exact address of the shellcode start is not always easy to guess

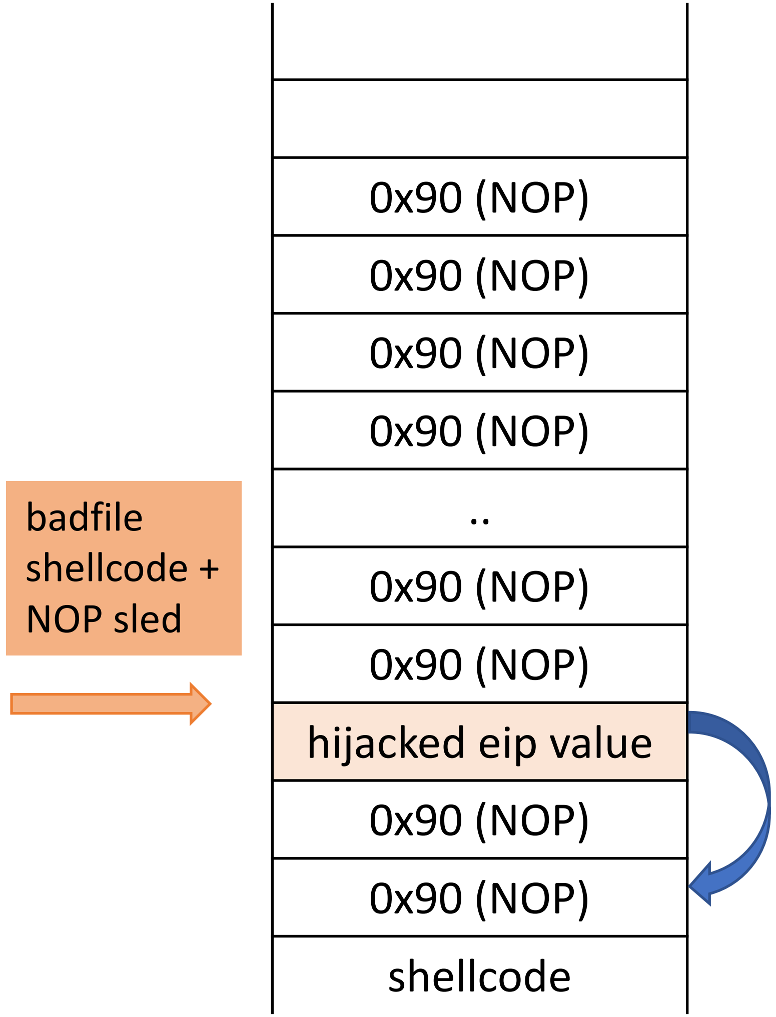
Miss? Segfault

Fix? NOP Sled!



NOP Sled!

- NOP instruction: 0x90
- NOP sleds are used to pad out exploits
  - Composed of instruction sequences that don't affect proper execution of the attack
  - Classically the NOP instruction (0x90), but not restricted to that
- Why are the called sleds?
  - Execution *slides* down the NOPs into your payload
  - Overwritten return address can be less precise, so long as we land somewhere in the NOP sled

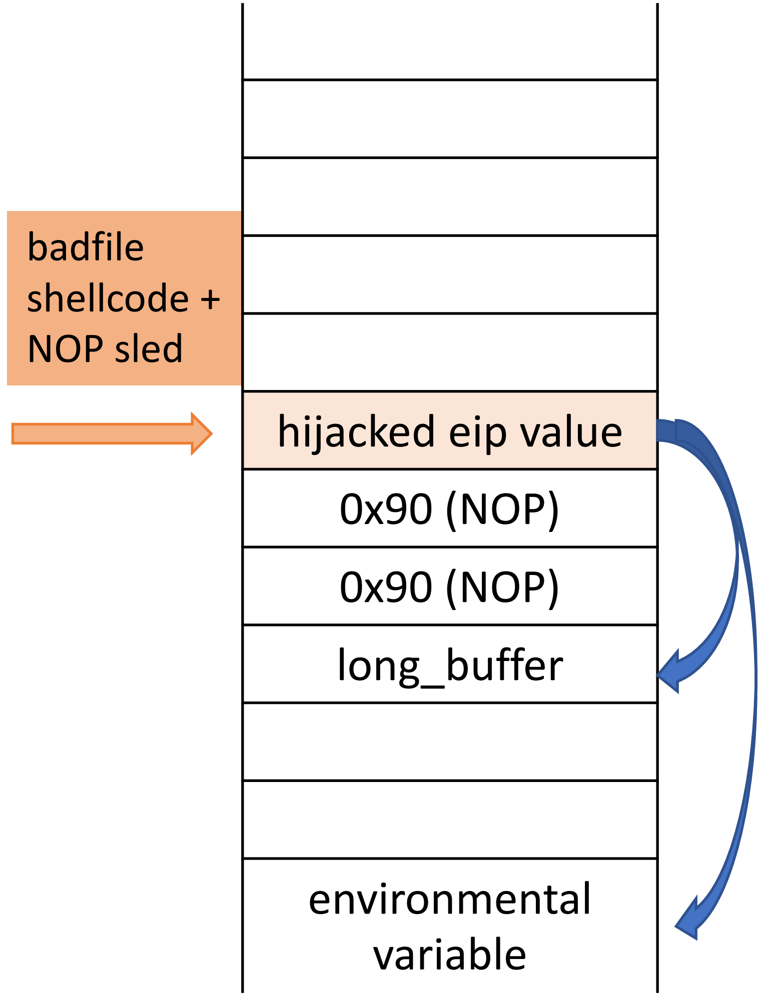


Small Buffers

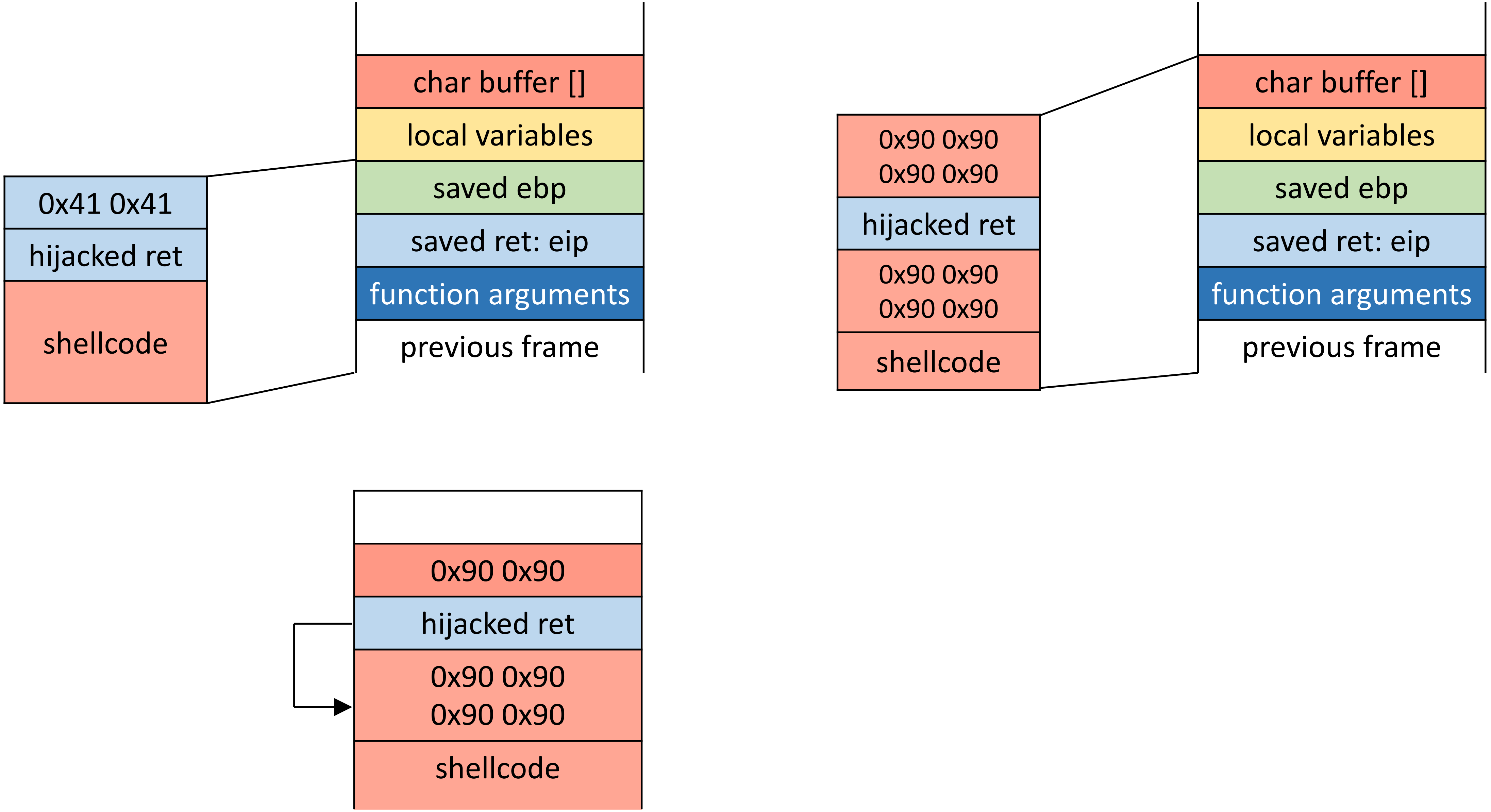
Buffer can be too small to hold exploit Code

Store exploit code in:

- an environmental variable
- or another buffer allocated on the stack
- redirect return address accordingly



Putting it all together



Summary: Stack Code Injection

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- For the basic stack-smashing attack, 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 crash with segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

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 <u>not</u> 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

```c
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
- 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,….)
```

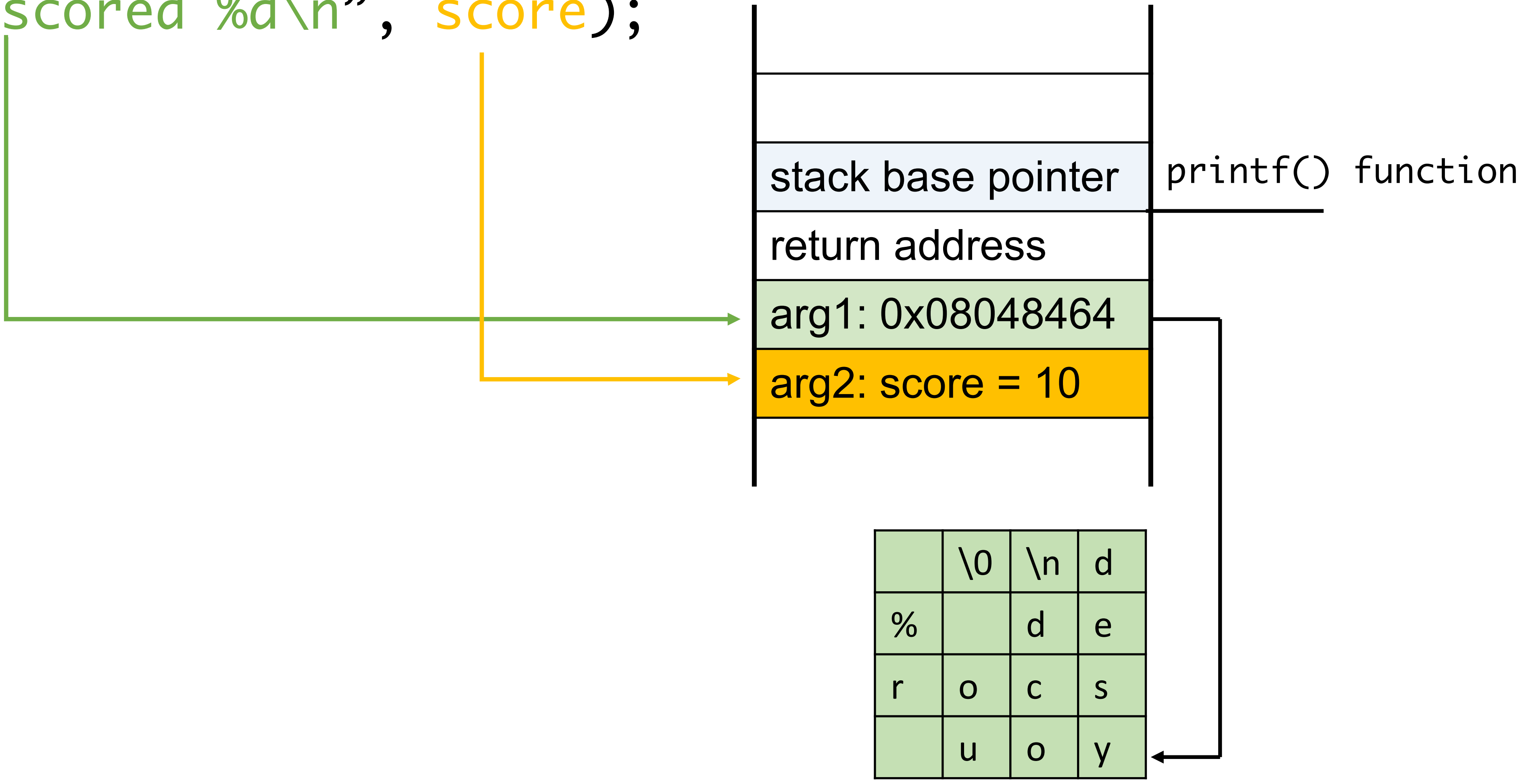
Usage:

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

format specification encoded by special % characters

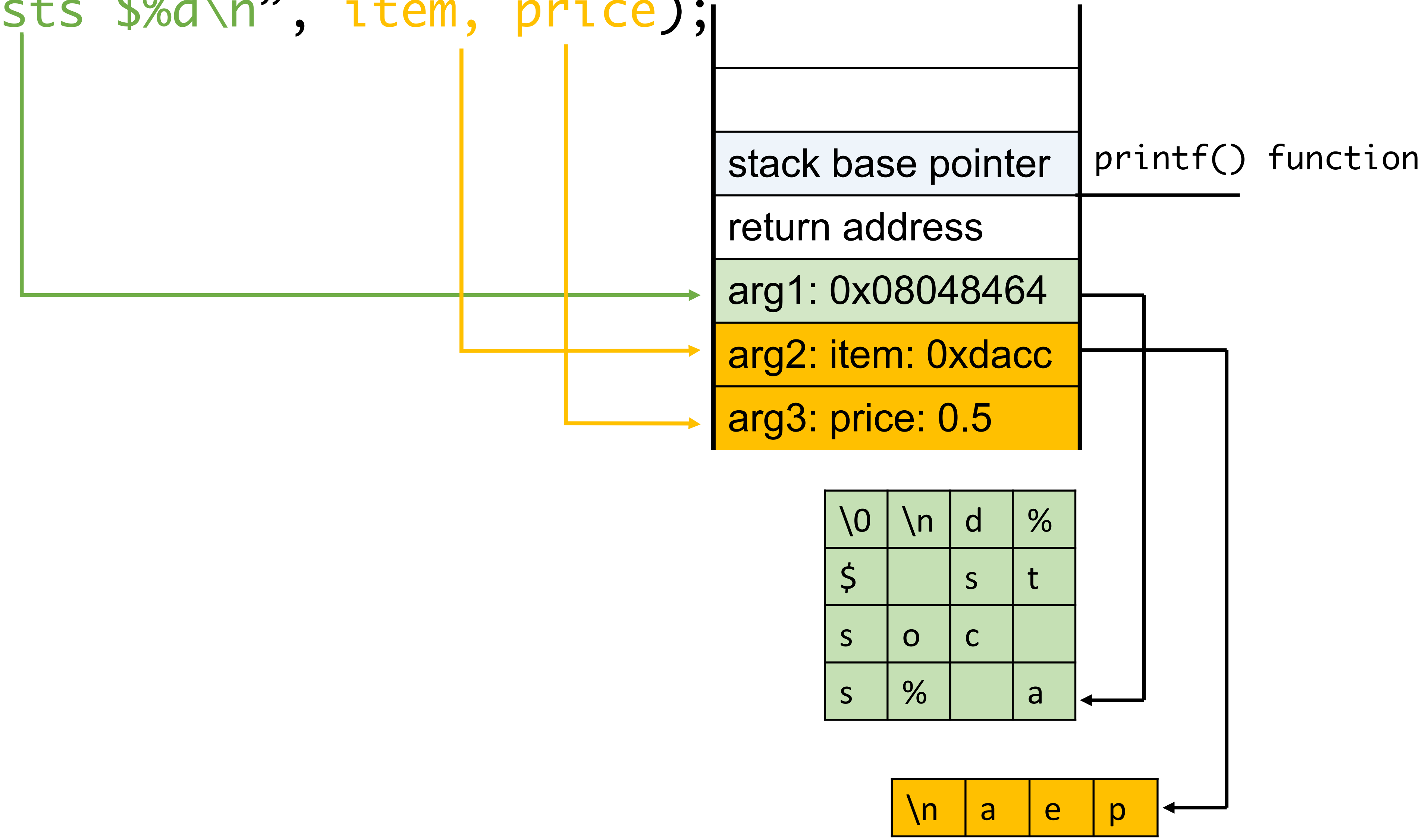
fun with format strings

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



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

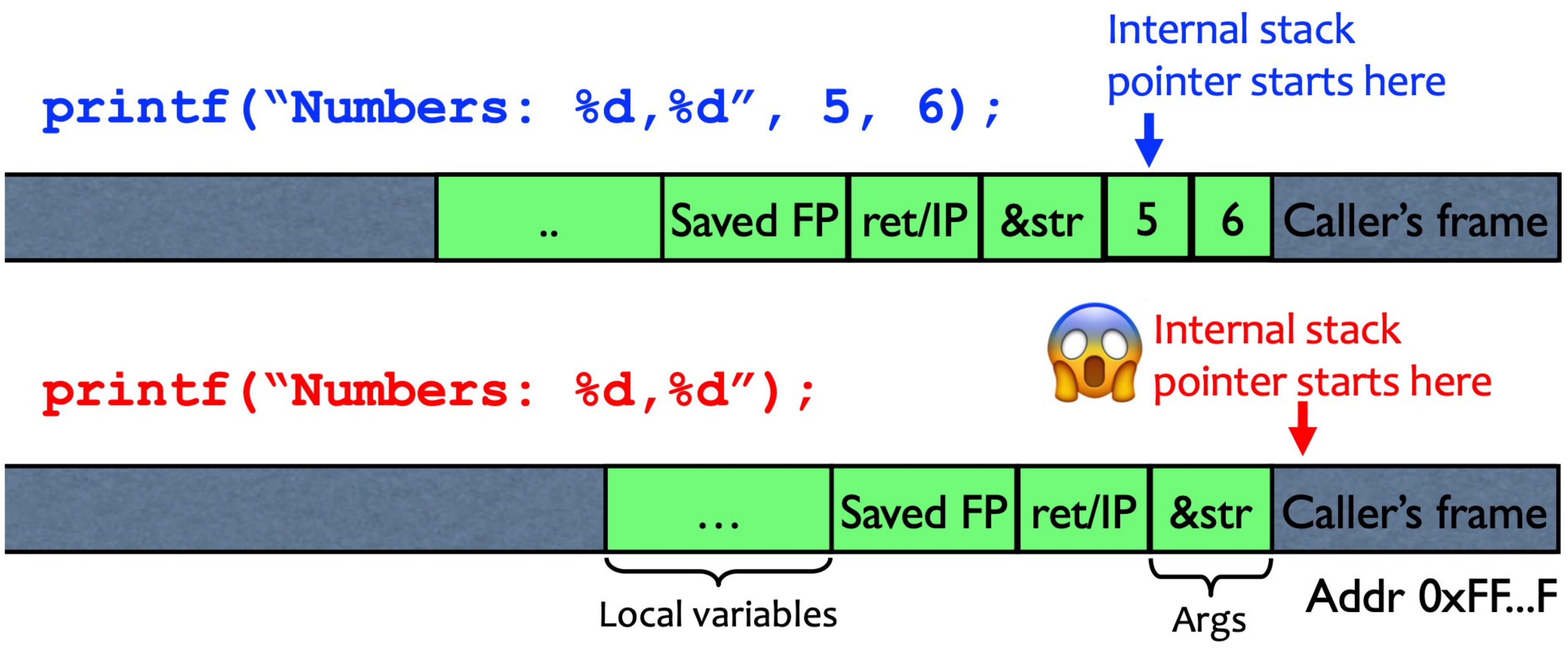
```
void printf(const char* format, ...)
{
      int i; char c; char* s; double d;
      va_list ap; /* declare an "argument pointer" to a variable arg list */
      va_start(ap, format); /* initialize arg pointer using last known arg */

      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

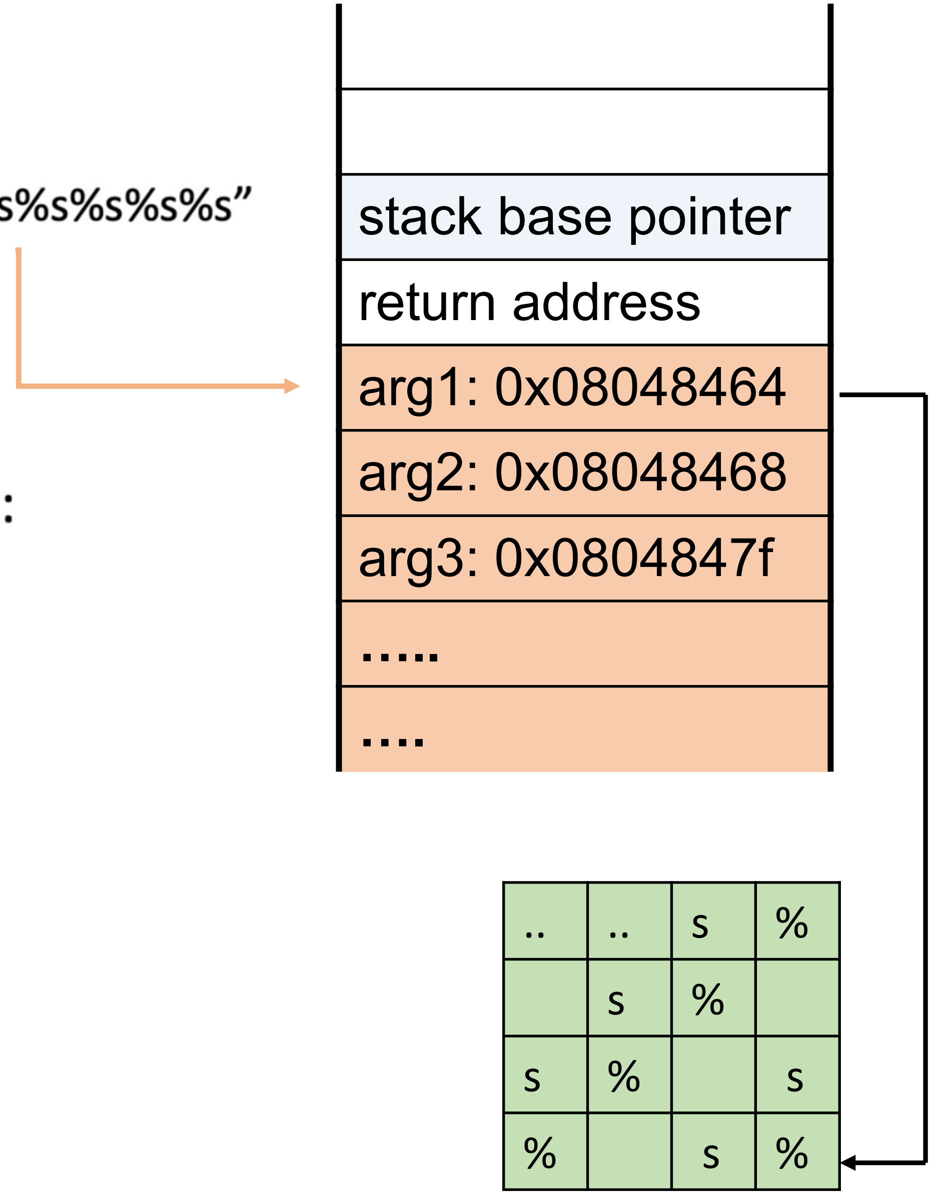
Closer look at the stack



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

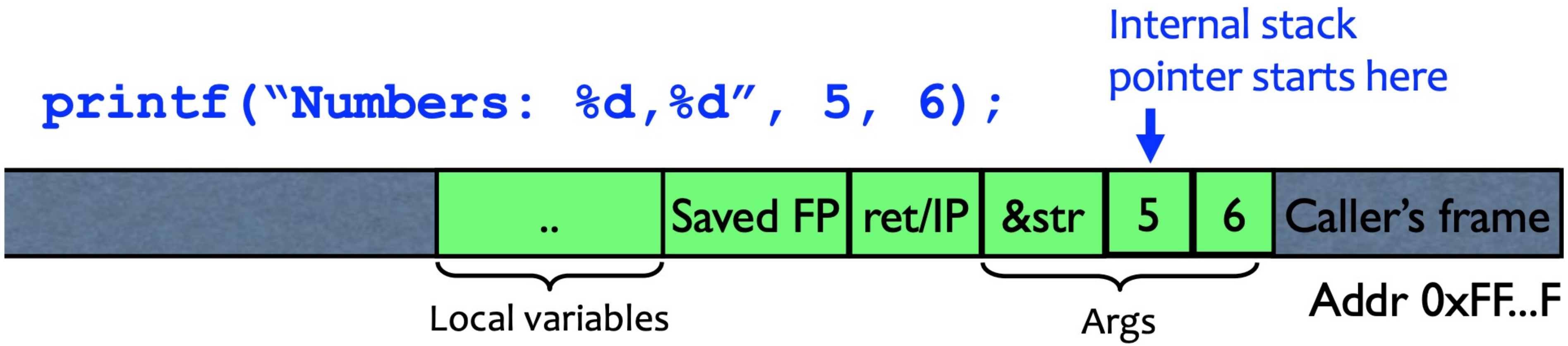
The %n format specifier

- `%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);`

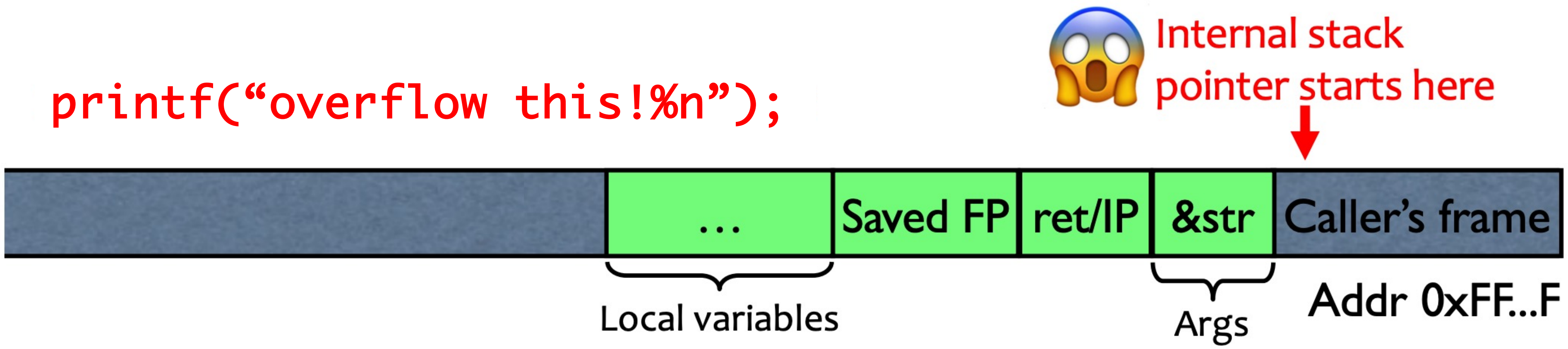
- Stack location pointed to by printf’s internal stack pointer will be interpreted as an address
- Write # characters at this address

Closer look at the stack

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




```
printf("overflow this!%n");
```



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");
```

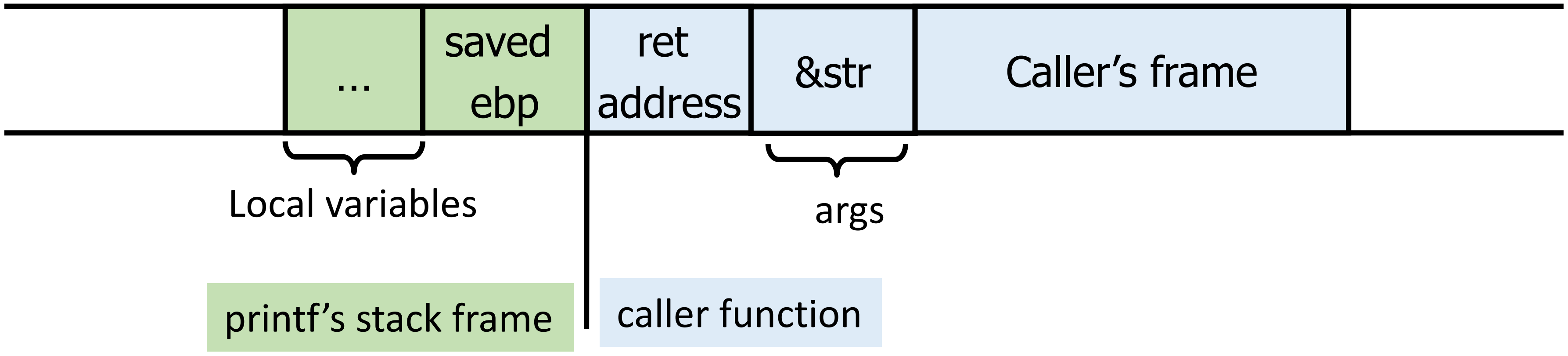
Output

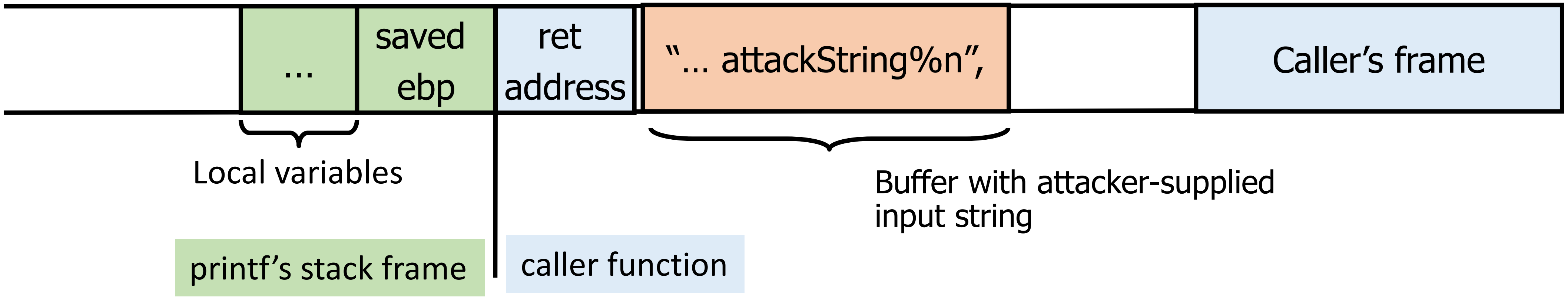
```
40012980.080628c4.bffff7a4.00000005.08059c04
```

instruct printf:

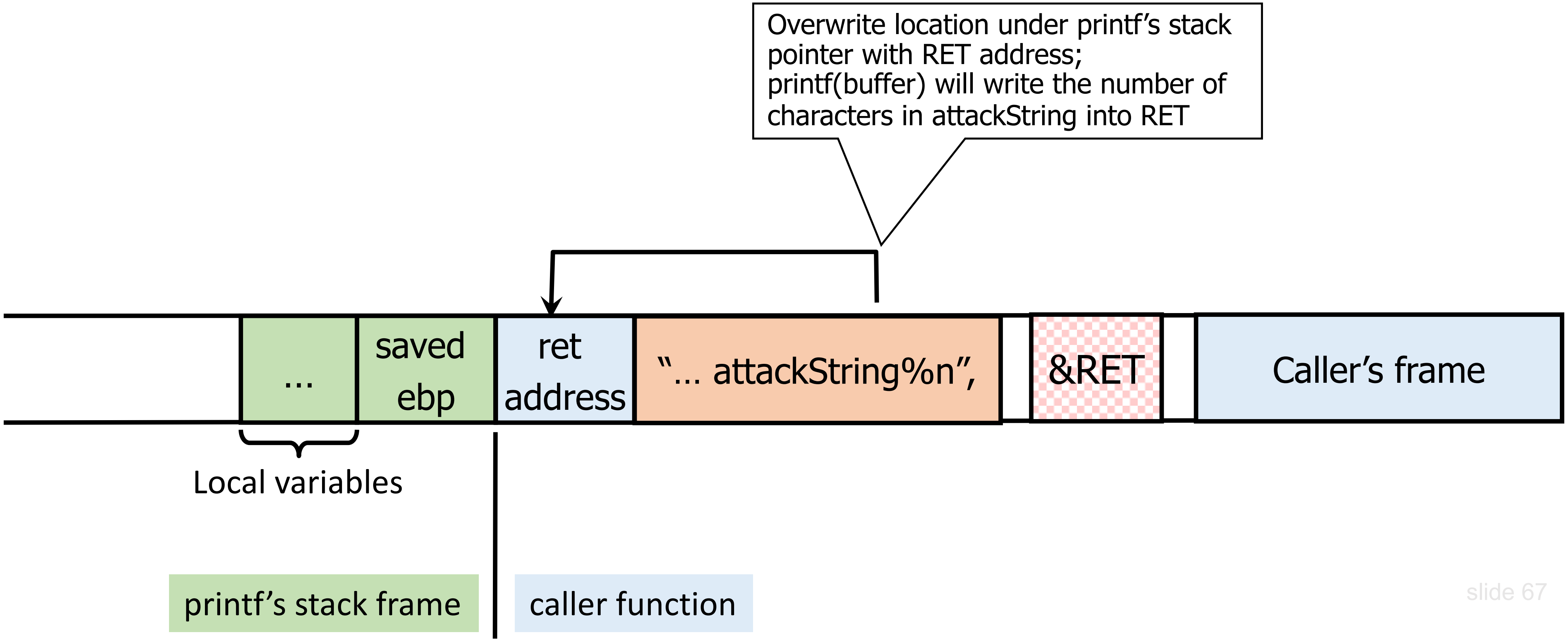
- retrieve 5 parameters
- display them as 8-digit padded hexademical numbers

Using %n to Mung Return Address

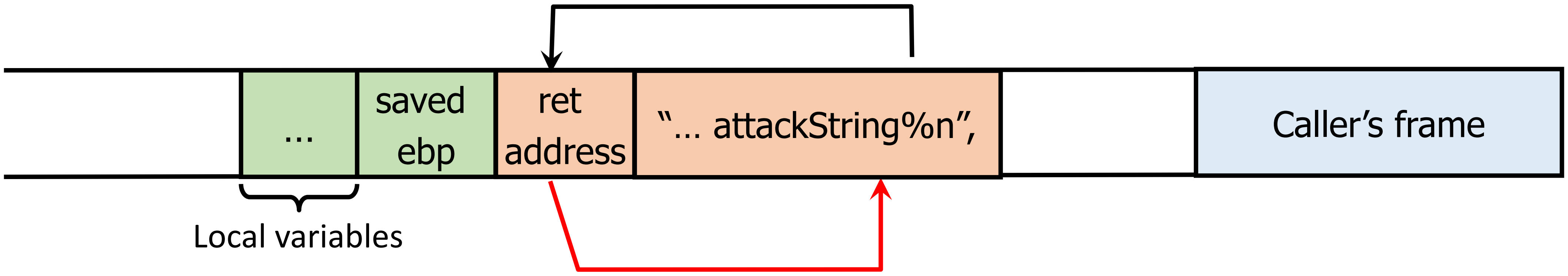




Using %n to Mung Return Address



Using %n to Mung Return Address



C has a concise way of printing multiple symbols:

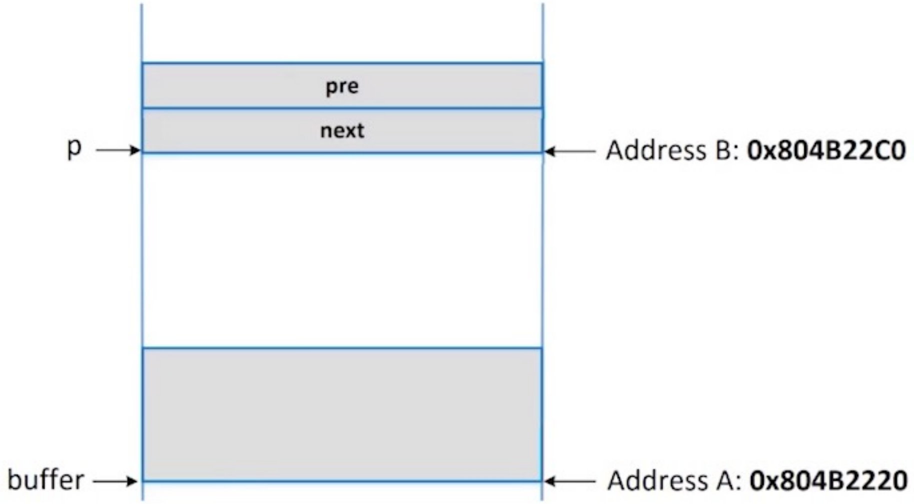
- %Mx will print exactly 4M bytes (taking them from the stack).
- Attack string should contain enough "%Mx" so that the number of characters printed is equal to the most significant byte of the address of the attack code.
- Repeat three times (four "%n" in total) to write into &RET+1, &RET+2, &RET+3,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

Heap Overflow

Heap based buffer overflow



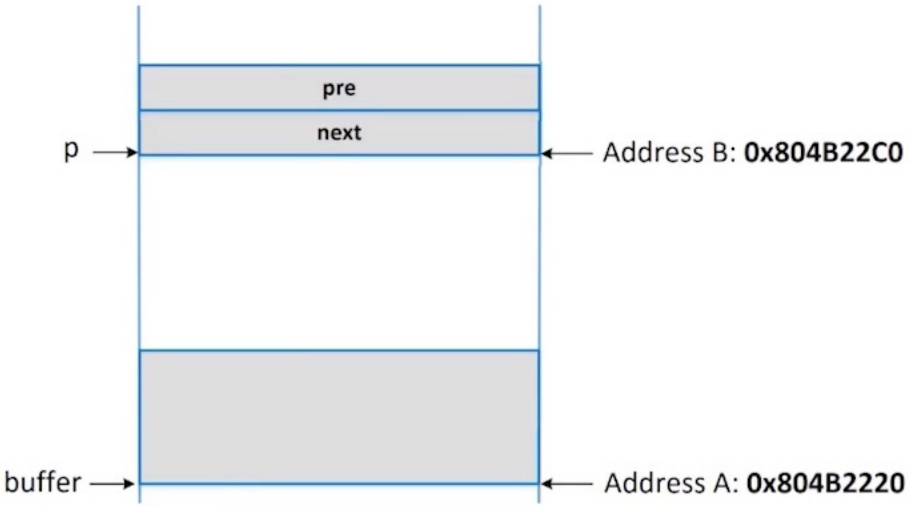

```
struct Node
{
  struct Node *next;
  struct Node *pre;
};
```

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

- Heap stores “chunks” of memory using linked 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…

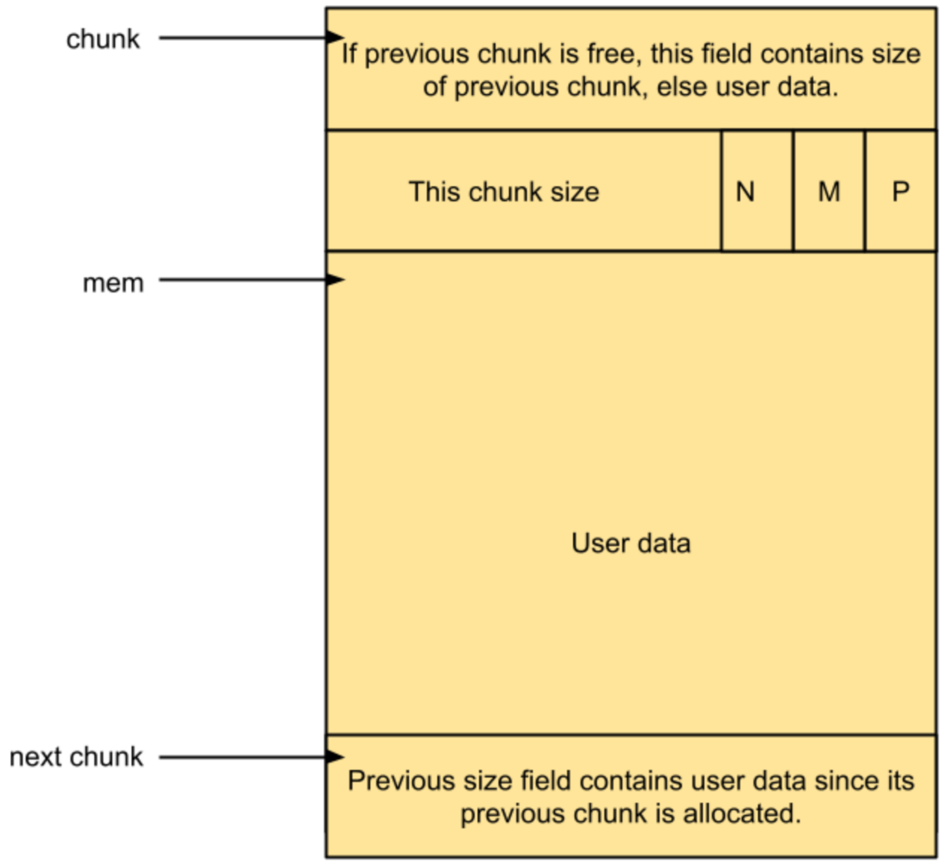
Heaps

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

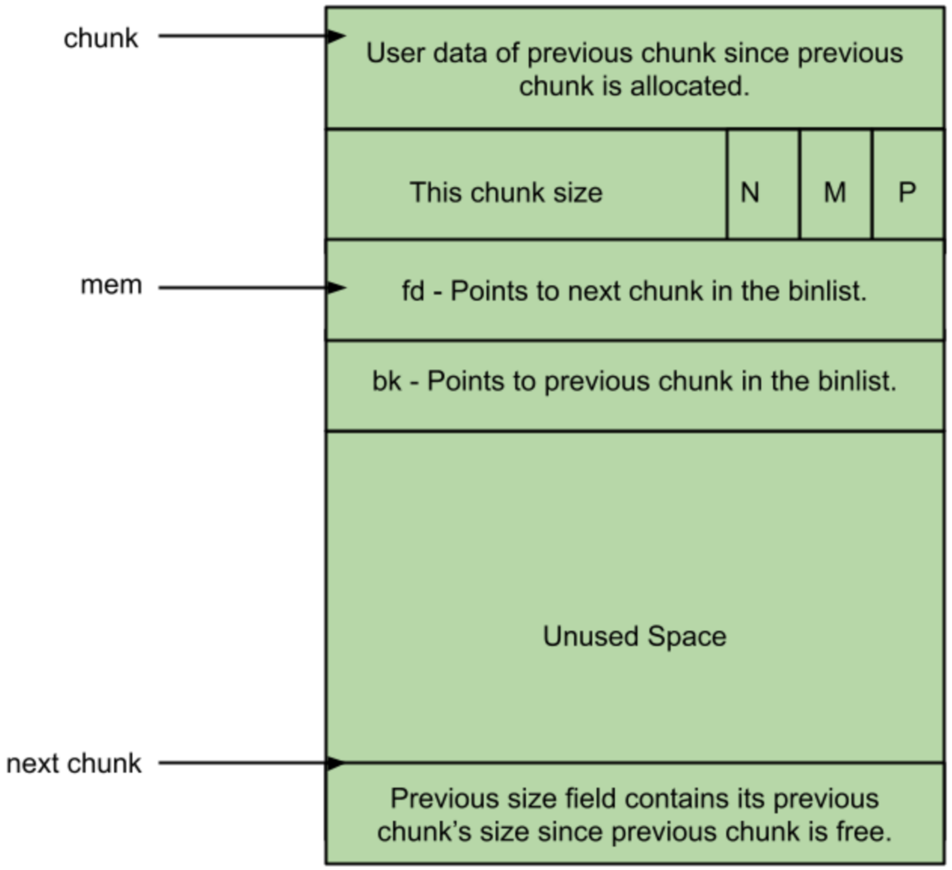
ptmalloc

- 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

ptmalloc: datastructures



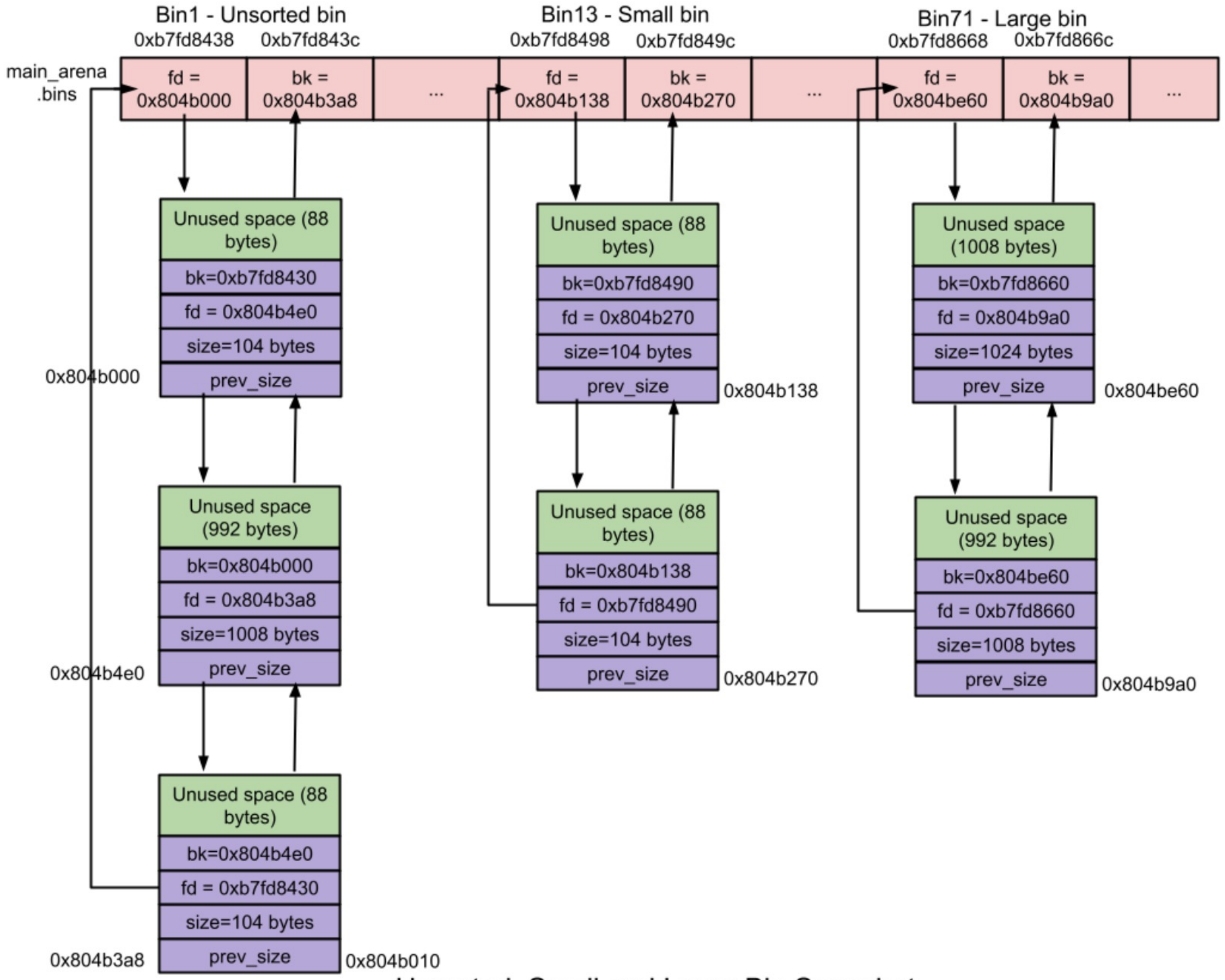
Allocated Chunk



Free Chunk

source: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

ptmalloc: datastructures



Unsorted, Small and Large Bin Snapshot

source: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

A summary of the attacks has been described below:

Attack	Target	Technique
First Fit	This is not an attack, it just demonstrates the nature of glibc's allocator	---
Double Free	Making `malloc` return an already allocated fastchunk	Disrupt the fastbin by freeing a chunk twice
Forging chunks	Making `malloc` return a nearly arbitrary pointer	Disrupting fastbin link structure
Unlink Exploit	Getting (nearly)arbitrary write access	Freeing a corrupted chunk and exploiting `unlink`
Shrinking Free Chunks	Making `malloc` return a chunk overlapping with an already allocated chunk	Corrupting a free chunk by decreasing its size
House of Spirit	Making `malloc` return a nearly arbitrary pointer	Forcing freeing of a crafted fake chunk
House of Lore	Making `malloc` return a nearly arbitrary pointer	Disrupting smallbin link structure
House of Force	Making `malloc` return a nearly arbitrary pointer	Overflowing into top chunk's header
House of Einherjar	Making `malloc` return a nearly arbitrary pointer	Overflowing a single byte into the next chunk

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

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/

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/