Program analysis application: memory safety

Gang Tan

CSE 597: Special topics on program analysis

Spring 2021

Penn State University

Why study this?

- Memory safety is an important program analysis application domain
  - Numerous dynamic/static analysis studied for identifying memory safety violations in C/C++ applications
- Some knowledge here needed for some of our homework (e.g., fuzzing)

Some Terminology

- Software **error**
  - A programming mistake that makes the software not meet its expectation
- Software **vulnerability**
  - A software error that can lead to possible attacks
- **Attack**
  - The process of exploiting a vulnerability
  - An attack can exploit a vulnerability to achieve additional functionalities for attackers
    - E.g., privilege escalation, arbitrary code execution

Language of Choice for Systems Programming: C/C++

- Systems software
  - OS; hypervisor; web servers; firmware; network controllers; device drivers; compilers; …
- Benefits of C/C++: programming model close to the machine model; flexible; efficient
- BUT **error-prone**
  - C/C++ not memory safe; huge security risk
  - Debugging memory errors is a headache
    - Perhaps on par with debugging multithreaded programs

Buffer Overflows

- Refer to reading/writing a buffer out of its bounds
  - Programmers' job in C/C++ to not do this
  - In contrast, many modern languages (Java, Python, …) prevent buffer overflows by performing automatic bounds checking
- The first Internet worm, and many subsequent ones (CodeRed, Blaster, ...), exploited buffer overflows
- Buffer overflows still cause many security alerts nowadays
  - E.g., check out CERT, cve.mitre.org, or bugtraq

C STRINGS: USAGE AND PITFALLS

C-Style Strings

- C-style strings consist of a contiguous sequence of characters, terminated by and including the first null character.
  - String length is the number of bytes preceding the null character.
  - The number of bytes required to store a string is the number of characters plus one (times the size of each character).

h	e	l	l	o	\0

Using Strings in C

- C provides many string functions in its libraries (libc)
- For example, we use the strcpy function to copy one string to another:

```
#include <string.h>
char string1[] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

Using Strings in C

- Another lets us compare strings

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
  printf("strings are equal\n");
else printf("strings are different\n")
```

- This code fragment will print "strings are different". Notice that strcmp does **not** return a boolean result.

Other Common String Functions

- strlen: getting the length of a string
- strcpy/strncpy: string copying
- strcat/strncat: string concatenation
- gets, fgets: receive input to a string
- …

Common String Manipulation Errors

- Programming with C-style strings, in C or C++, is error prone
- Common errors include
  - Buffer overflows
  - null-termination errors
  - off-by-one errors
  - …

gets: Unbounded String Copies

- Occur when data is copied from an unbounded source to a fixed-length character array

```
void main(void) {
  char Password[8];
  puts("Enter a 8-character password:");
  gets(Password);
  printf("Password=%s\n",Password);
}
```

strcpy and strcat

- The standard string library functions do not know the size of the destination buffer

```
int main(int argc, char *argv[]) {
        char name[2048];
        strcpy(name, argv[1]);
        strcat(name, " = ");
        strcat(name, argv[2]);
        ...
}
```

Better String Library Functions

- Functions that restrict the number of bytes are often recommended
- Never use gets(buf)
  - Use `fgets(buf, size, stdin)` instead

From gets to fgets

- char *fgets(char *BUF, int N, FILE *FP);
  - *“Reads at most N-1 characters from FP until a newline is found. The characters including to the newline are stored in BUF. The buffer is terminated with a 0.”*

void main(void) {
    char Password[~~8~~ 9];
    puts("Enter a 8-character password:");
    fgets(Password, ~~8~~ 9, stdin);
    ...
}

Better String Library Functions

- Instead of strcpy(), use strncpy()
- Instead of strcat(), use strncat()
- Instead of sprintf(), use snprintf()

But Still Need Care

- char *strncpy(char *s1, const char *s2, size_t n);
  - *“Copy not more than n characters (including the null character) from the array pointed to by s2 to the array pointed to by s1; If the string pointed to by s2 is shorter than n characters, null characters are appended to the destination array until a total of n characters have been written.”*
  - What happens if the size of s2 is n or greater
    - It gets truncated
    - **And s1 may not be null-terminated!**

Null-Termination Errors

```
int main(int argc, char* argv[]) {
    char a[16], b[16];
    strncpy(a, "0123456789abcdef", sizeof(a));
    printf(“%s\n”,a);
    strcpy(b, a);
}
```

a[] not properly terminated. Possible segmentation fault if printf(“%s\n”,a);

How to fix it?

strcpy to strncpy

- Don't replace

  strcpy(dest, src)

  by

  strncpy(dest, src, sizeof(dest))

  but by

  strncpy(dest, src, sizeof(dest)-1)

  dst[sizeof(dest)-1] = `\0`;

  if dest should be null-terminated!

- You never have this headache in memory-safe languages
- Further, strncpy has big performance penalty vs. strcpy
  - It NIL-fills the remainder of the destination

Signed vs Unsigned Numbers

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N)
  {error ("invalid length"); return; }
read(fd, buf, len);
```

We forget to check for negative lengths

`len` cast to unsigned and negative length overflows

*slide by Eric Poll

Checking for Negative Lengths

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N || len < 0)
   {error (“invalid length"); return; }
read(fd, buf, len);
```

It still has a problem
if the buf is going to be treated as a C string.

*slide by Eric Poll

A Good Version

```
char buf[N];
int i, len;

read(fd, &len, sizeof(len));
if (len > N-1 || len < 0)
   {error (“invalid length"); return; }
read(fd, buf, len);
buf[len] = '\0'; // null terminate buf
```

*slide by Eric Poll

Integer Overflow Vulnerabilities

* slides adapted from those by Seacord

Integer Overflows

- An **integer overflow** occurs when an integer is increased beyond its maximum value or decreased beyond its minimum value
- Standard integer types (signed)
  - signed char, short int, int, long int, long long int
- Signed overflow vs unsigned overflow
  - An unsigned overflow occurs when the underlying representation can no longer represent an integer value.
  - A signed overflow occurs when a value is carried over to the sign bit

Overflow Examples

```
unsigned int ui;
signed int si;
ui = UINT_MAX; // 4,294,967,295;
ui++;
printf("ui = %u\n", ui);

si = INT_MAX; // 2,147,483,647
si++;
printf("si = %d\n", si);
```

ui = 0

si = -2,147,483,648

Overflow Examples, cont'd

```
ui = 0;
ui--;
printf("ui = %u\n", ui);
```

ui = 4,294,967,295

```
si = INT_MIN; // -2,147,483,648;
si--;
printf("si = %d\n", si);
```

si = 2,147,483,647

Integer Overflow Example

```
int main(int argc, char *const *argv) {
    unsigned short int total;
    total = strlen(argv[1]) + strlen(argv[2]) + 1;
    char *buff = (char *) malloc(total);
    strcpy(buff, argv[1]);
    strcat(buff, argv[2]);
}
```

What if the total variable is overflowed because of the addition operation?

Vulnerability: JPEG Example

- Based on a real-world vulnerability in the handling of the comment field in JPEG files

```
void getComment(unsigned int len, char *src) {
    unsigned int size;
    size = len - 2;
    char *comment = (char *)malloc(size + 1);
    memcpy(comment, src, size);
    return;
}
```

size is interpreted as a large positive value of 0xffffffff

size+1 is 0

What if I do “getComment(1, "Comment ");”?

Possible to cause an overflow by creating an image with a comment length field of 1

Vulnerability: Truncation Errors

```
int func(char *name, long cbBuf) {
    unsigned short bufSize = cbBuf;
    char *buf = (char *)malloc(bufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        …
        free(buf);
        return 0;
    }
    return 1;
}
```

What if we call the function with cbBuf greater than 2^{16} -1?

Heap Overflow

*adapted from slides by Trent Jaeger

Heap Overflows

- Stack overflow: overflow a memory region on the stack
- Heap overflow: overflow a memory region dynamically allocated on the heap

```
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
  PacketRead(packet);
  if (Authenticate(packet))
    authenticated = 1;
}
if (authenticated)
 ProcessPacket(packet);
```

What happens if PacketRead overflows the packet buffer and overwrite important data in memory?

Overflowing Heap Critical User Data

```
/* record type to allocate on heap */
typedef struct chunk {
    char inp[64];              /* vulnerable input buffer */
    void (*process)(char *);  /* pointer to function */
} chunk t;

void showlen(char *buf) {
    int len; len = strlen(buf);
    printf("buffer5 read %d chars\n", len);
}

int main(int argc, char *argv[]) {
    chunk t *next;
    setbuf(stdin, NULL);
    next = malloc(sizeof(chunk t));
    next->process = showlen;
    printf("Enter value: ");
    gets(next->inp);
    next->process(next->inp);
    printf("buffer5 done\n");
}
```

example by Stallings

- Overflow the buffer on the heap so that the function pointer is changed to an arbitrary address

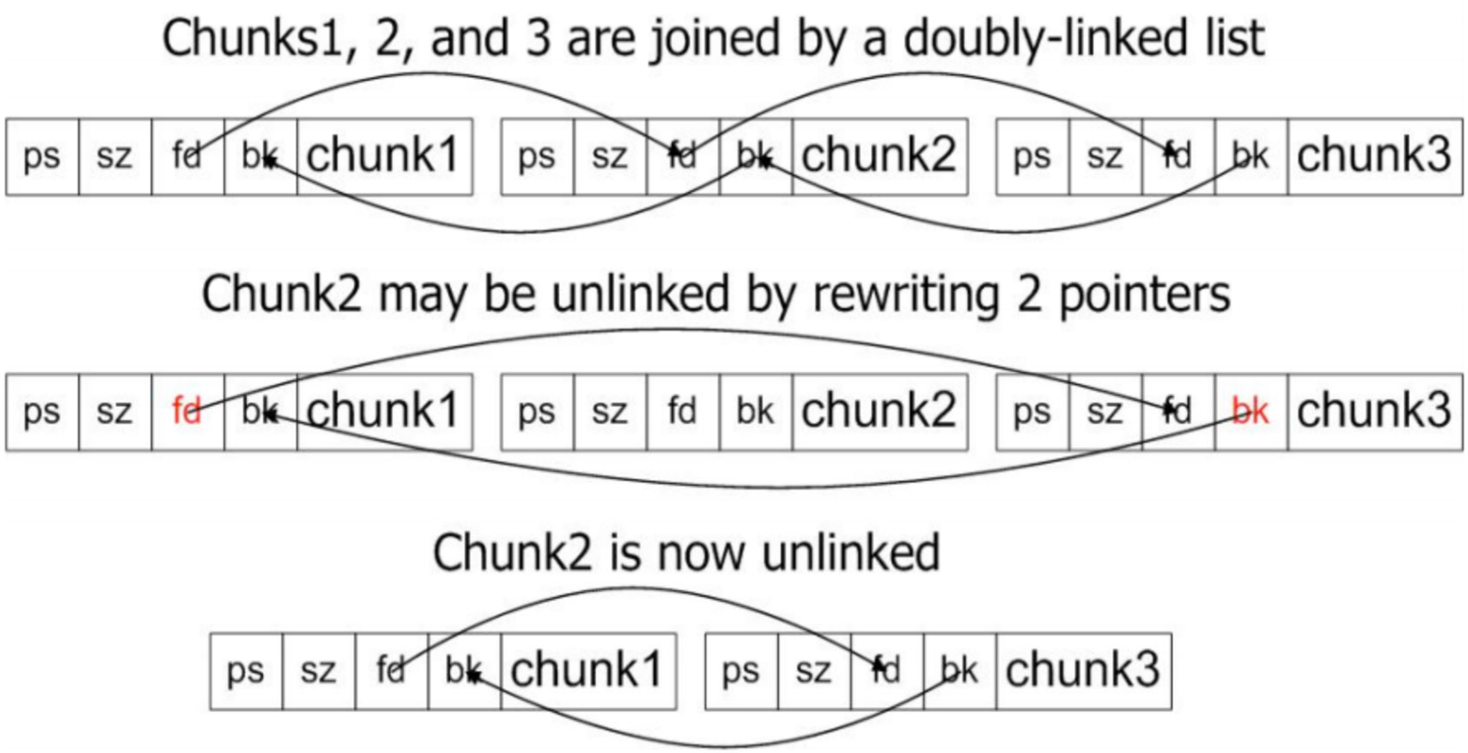
Overflow Heap Meta-Data

- Heap allocators (AKA memory managers)
  - What regions have been allocated and their sizes
  - What regions are available for allocation
- Heap allocators maintain metadata such as chunk size, previous, and next pointers
  - Metadata adjusted during heap-management functions
    - malloc() and free()
  - Heap metadata often adjacent to heap user data

Example Heap Allocator

- Maintain a doubly-linked list of allocated and free chunks
- malloc() and free() modify this list

An Example of Removing a Chunk



- free() removes a chunk from allocated list
  - chunk2->bk->fd = chunk2->fd
  - chunk2->fd->bk = chunk2->bk

Attacking the Example Heap Allocator

- By overflowing chunk2, attacker controls bk and fd of chunk2
- Suppose the attacker wants to write value to memory address addr
  - Attacker sets chunk2->fd to be value
  - Attacker sets chunk2->bk to be addr-offset, where offset is the offset of the fd field in the structure

Attacking the Example Heap Allocator

- free() changed in the following way
  - chunk2->bk->fd = chunk2->fd becomes (addr-offset)->fd = value, the same as (*addr)=value
  - chunk2->fd->bk= chunk2->bk becomes value->bk = addr-offset
- The first memory write achieves the attacker's goal
  - Arbitrary memory writes

Use After Free and Double Free

*adapted from slides by Trent Jaeger

Use After Free

- Error: Program frees memory on the heap, but then references that memory as if it were still valid
  - Adversary can control data written using the freed pointer
- AKA use of dangling pointers

Use After Free

```
int main(int argc, char **argv) {
   char *buf1, *buf2, *buf3;

   buf1 = (char *) malloc(BUFSIZE1);

   free(buf1);

   buf2 = (char *) malloc(BUFSIZE2);
   buf3 = (char *) malloc(BUFSIZE2);
   strncpy(buf1, argv[1], BUFSIZE1-1);
   …

}
```

What happens here?

Use After Free

- When the first buffer is freed, that memory is available for reuse right away
- Then, the following buffers are possibly allocated within that memory region

```
buf2 = (char *) malloc(BUFSIZE2);
buf3 = (char *) malloc(BUFSIZE2);
```

- Finally, the write using the freed pointer may overwrite buf2 and buf3 (and their metadata)

```
strncpy(buf1, argv[1], BUFSIZE1-1);
```

Use After Free

- Most effective attacks exploit data of another type

```
struct A {
    void (*fnptr)(char *arg);
    char *buf;
};

struct B {
    int B1;
    int B2;
    char info[32];
};
```

Use After Free

- Free A, and allocate B does what?

```
x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));
```

Use After Free

- How can you exploit it?

```
x = (struct A *)malloc(sizeof(struct A));
free(x);
y = (struct B *)malloc(sizeof(struct B));

y->B1 = 0xDEADBEEF;
x->fnptr(x->buf);
```

- Assume that
  - The attacker controls what to write to y->B1
  - There is a later use-after-free that performs a call using "x->fnptr"
- Become a popular vulnerability to exploit – over 60% of CVEs in 2018

Exercise: Find the Use-After-Free Error

```
#include <stdlib.h>

struct node {
  struct node *next;
};

void func(struct node *head) {
  struct node *p;
  for (p = head; p != NULL; p = p->next) {
    free(p);
  }
}
```

Prevent Use After Free

- Difficult to detect because these often occur in complex runtime states
  – Allocate in one function
  – Free in another function
  – Use in a third function
- It is not fun to check source code for all possible pointers
  – Are all uses accessing valid (not freed) references?
  – In all possible runtime states

Prevent Use After Free

- What can you do that is not too complex?
  - You can set all freed pointers to NULL
    - Getting a null-pointer dereference if using it
    - Nowadays, OS has built-in defense for null-pointer deference
  - Then, no one can use them after they are freed
  - Complexity: need to set all aliased pointers to NULL

Related Problem: Double Free

```
main(int argc, char **argv)
{
        …
        buf1 = (char *) malloc(BUFSIZE1);
        free(buf1);
        buf2 = (char *) malloc(BUFSIZE2);
        strncpy(buf2, argv[1], BUFSIZE2-1);
        free(buf1);
        free(buf2);
}
```

What happens here?

Double Free

- Free buf1, then allocate buf2
  - buf2 may occupy the same memory space of buf1
- buf2 gets user-supplied data

  ```
  strncpy(buf2, argv[1], BUFSIZE2-1);
  ```
- Free buf1 again
  - Which may use some buf2 data as metadata
  - And may mess up buf2's metadata
- Then free buf2, which uses really messed up metadata

What’s Wrong? Fix?

```
#include <stdlib.h>

int f(size_t n) {
  int error_condition = 0;

  int *x = (int *)malloc(n * sizeof(int));
  if (x == NULL)
    return -1;

  /* Use x and set error_condition on error. */
  …

  if (error_condition == 1) {
    /* Handle error */
    free(x);
  }

  free(x);
  return error_condition;
}
```

What’s Wrong? Fix?

```
#include <stdlib.h>

/* p is a pointer to dynamically allocated memory. */
void func(void *p, size_t size) {
  p2 = realloc(p, size);
  if (p2 == NULL) {
    free(p);
    return;
  }
}
```

When size == 0,
realloc(p,0) same
As free(p)

Double Free

- So, “double free” can achieve the same effect as some heap overflow vulnerabilities
  - So, can be addressed in the same way
  - But, you can also save yourself some headache by setting freed pointers to NULL
  - Some new heap allocators nowadays have built-in defense against double free

Format String Attacks

Format String Attacks

- Public since 1999
  - first thought as harmless programming errors
- Format string refers to the argument that specifies the format of a string to functions like printf
- Example

```
int i;
printf (“i = %d with address %08x \n", i, &i);
```

format string

56

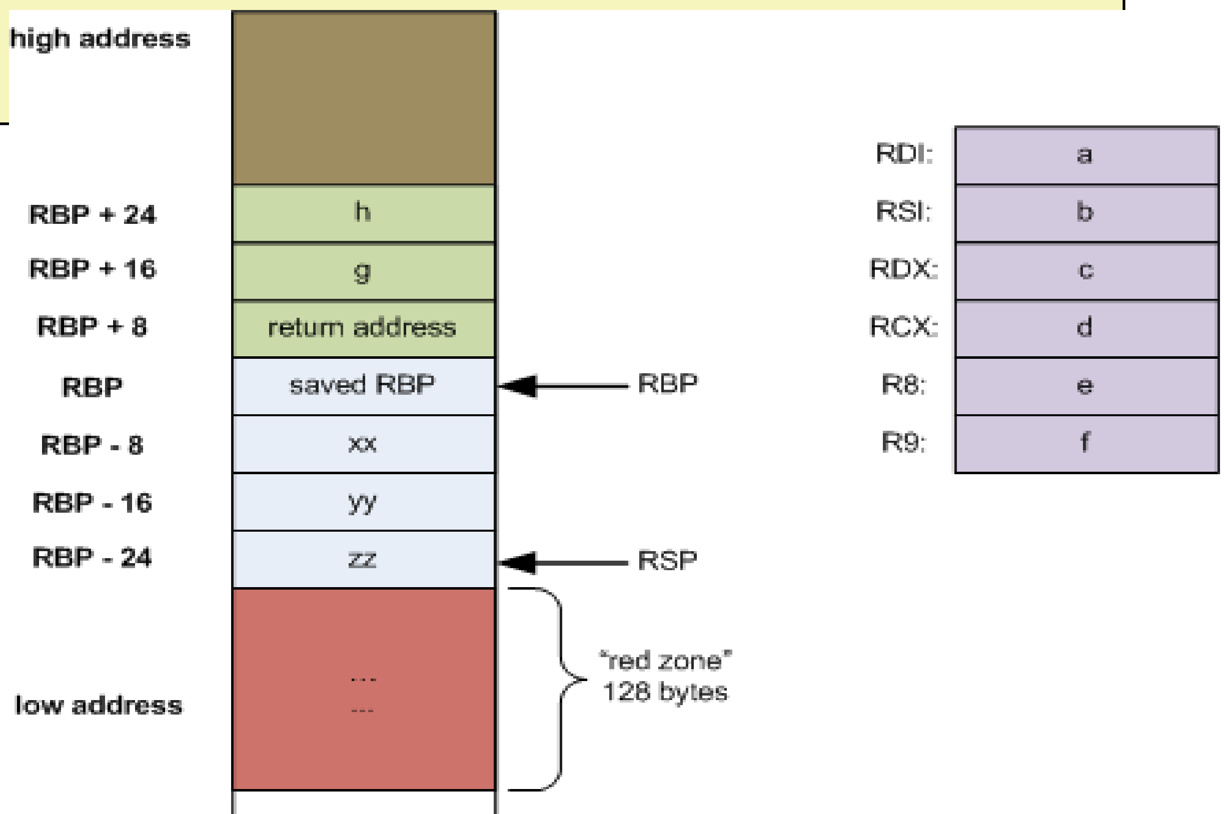
Argument Passing in x86-64

- Arguments (up to the first six) are passed to procedures via registers
  - The rest passed through the stack

Operand size (bits)	Argument Number 1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Example of Argument Passing

```c
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h) {
   long xx = a * b * c * d * e * f * g * h;
   long yy = a + b + c + d + e + f + g + h;
   long zz = utilfunc(xx, yy, xx % yy);
   return zz + 20;
}
```



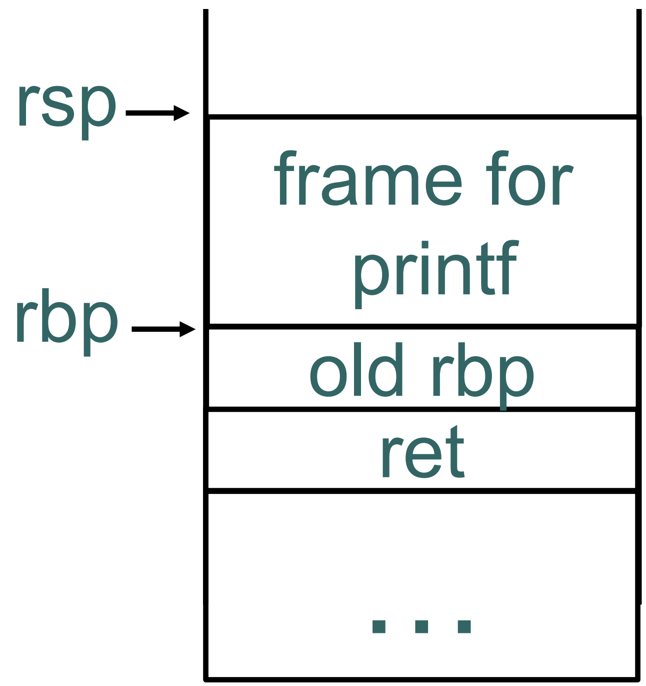
* Example from https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/

How Does printf Work in C?

int i;

printf ("i = %d with address %08x\n", i, &i);

- Pass string addr pointer, i, &i through registers rdi, rsi, rdx, and invokes printf
- When control is inside printf, the function looks for arguments in those registers
  - Or on the stack if more than 6 arguments



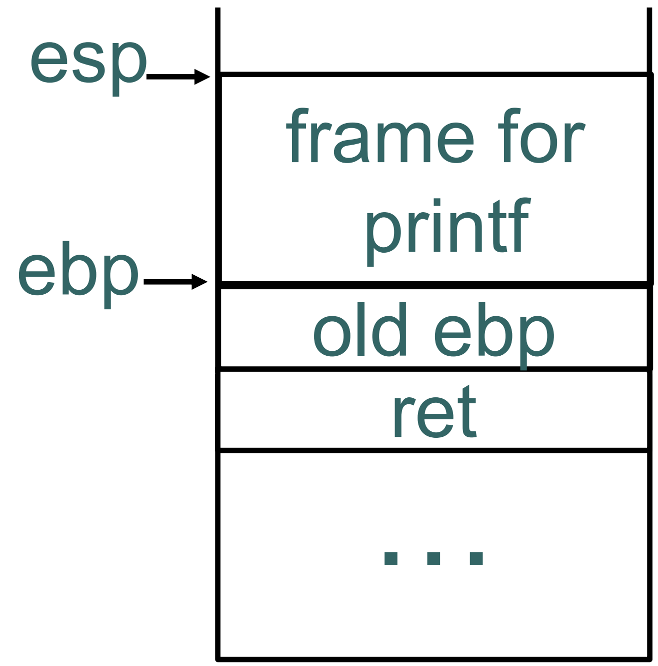
How Does printf Work in C?

- What happens for the following program?

  int i;

  printf (“i = %d with address %08x\n");

- The C compiler would not complain
  - pretending that the required arguments were at the right place



Format String Attacks

- What about the following simple program for echoing user input?

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

- Appears to be safe
- However, what would happen if the input is
  - hello%d%d%d%d%d%d
  - Essentially, it runs printf(“hello%d%d%d%d%d%d”)
    - printf with only one format-string argument
  - It would print numbers from argument-passing regs and the stack
  - An attacker can view memory this way
- What if the arg[1] is “hello%s”?
  - Likely segmentation fault (denial of service)

Format String Attacks

- Getting fancier
  - There is a “%n” specifier, which writes the number of bytes already printed, into a variable of our choice.

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

  - i gets 6 at the end

Format String Attacks

- For the echo program, what if the user input is "foobar%n"?
  - It will take an address from rsi, and write 6 to the memory slot with that address
  - What about "foobar%10u%n"?
    - Write possibly 16 to a memory location
  - How to write to an arbitrary address?
    - Put that address at the right place (in the right register or stack slot)
- Therefore, an attacker can possibly update any memory with arbitrary contents
  - How about overwriting a function pointer and hijacking the control flow (and installing some worm code)?

Format String Attacks

```
int main(int argc, char *argv[]) {
    char buf[512];
    fgets(buf, sizeof(buf), stdin);


    printf("The input is:");
    printf(buf);
    return 0;
}
```

No buffer overblow here

But, format string attacks

- An attacker can
  - view/change any part of the memory
  - execute arbitrary code
    - Just put the code into buf
- More details see paper
  - "Exploiting Format String Vulnerabilities"

Format String Attacks: Fixes

- Most of time: quite easy to fix

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

~~printf(argv[1]);~~

printf(“%s”, argv[1])

- But not always so obvious
  - sometimes not easy to find

```
void foo(char *user) {
    char outbuf[512];
    char buffer[512];
    sprintf (buffer, "ERR Wrong command: %400s", user);
    sprintf (outbuf, buffer);
}
```

Is there a buffer overflow? Is there a forma-string vulnerability?
How to fix it?

Prevent Format String Vulnerabilities

- Preventing format string vulnerabilities means limiting the ability of adversaries to control the format string
  - Hard-coded strings w/ no arguments – when you can
  - Hard-coded format strings at least – no printf(arg)
  - Do not use %n
    - Be careful with other references - %s and sprintf can be used to created disclosure attacks
  - Compiler support to match printf arguments with format string

Summary: Memory-Corruption Vulnerabilities

- Buffer overflow
  - stack smashing: overwrite data on the stack
  - can also overwrite data on the heap
  - Integer overflow makes it easier
- Use after free; double free
- Format string attacks
- And there are many more …
  - E.g., Mark Dowd showed it was possible to hijack Adobe Flash player based on a null pointer dereference
- Type-safe languages have other kinds of vulnerabilities
  - E.g., SQL injection attacks, …