Buffer Overflows

Software Security

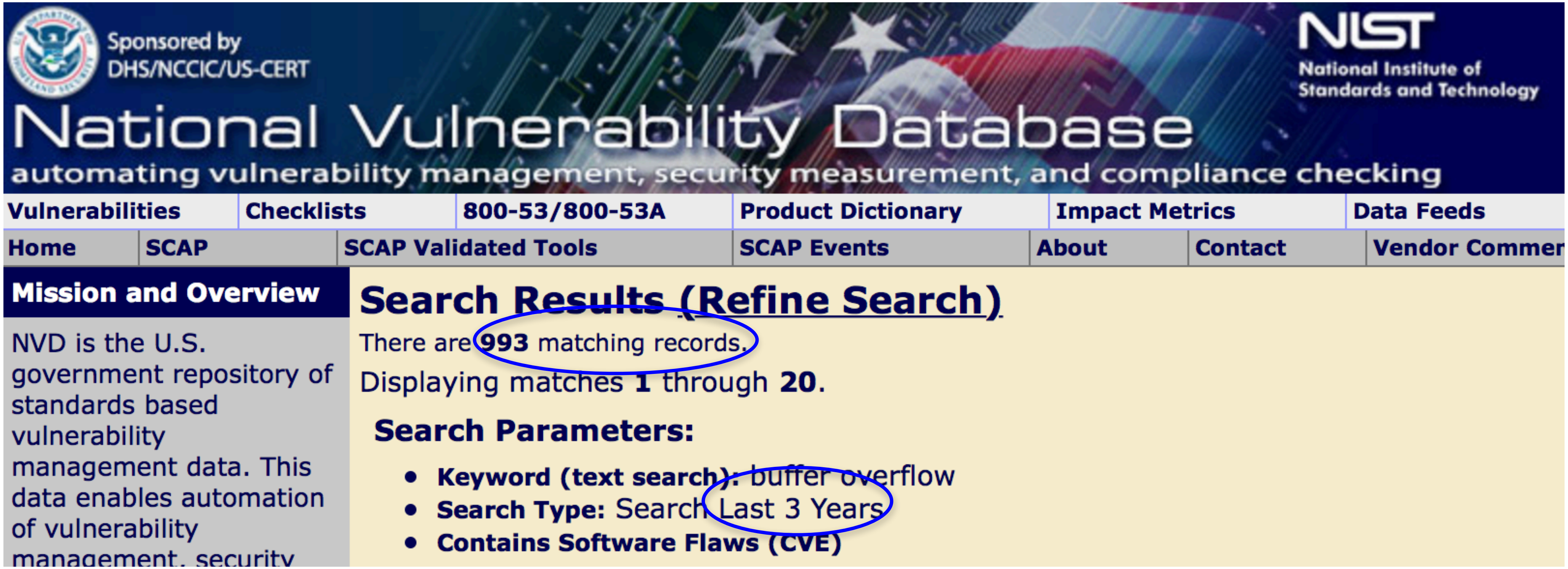
Pedro Adão 2022/23

(with Ana Matos & Miguel Pupo Correia)

TÉCNICO LISBOA

Book: Chapter 3 (v1) / 5 (v2)

Motivation


Sponsored by
DHS/NCCIC/US-CERT
NIST
National Institute of Standards and Technology
National Vulnerability Database
automating vulnerability management, security measurement, and compliance checking
Vulnerabilities
Checklists
800-53/800-53A
Product Dictionary
Impact Metrics
Data Feeds
Home
SCAP
SCAP Validated Tools
SCAP Events
About
Contact
Vendor Commer
Mission and Overview
NVD is the U.S. government repository of standards based vulnerability management data. This data enables automation of vulnerability
Search Results (Refine Search)
There are 993 matching records.
Displaying matches 1 through 20.
Search Parameters:
Keyword (text search): buffer overflow
Search Type: Search Last 3 Years
Contains Software Flaws (CVE)

Motivation

A newly discovered vulnerability in Microsoft's Outlook and Outlook Express programs leave thousands of computers open to attack from malicious email (...)

The bug is a classic "*buffer overflow*" error in the section of Outlook that parses the Date field of each incoming email. By padding the date with a long string of characters, an attacker can escape from the area of memory reserved for storing it, and into a section that executes instructions. From there, the attacker's email could secretly infect a victim computer with a "back door" program (…)

– Kevin Poulsen, MS Battles Outlook Bug, Security Focus, 19 de Julho de 2000

Cause

- C and C++:
  - Do not force the verification if data overflows the limit of a buffer / array / vector
  - e.g., because programmers make wrong assumptions like *the user never types more than 1000 characters as input*
- Several contributing factors:
  - Large number of unsafe string operations
    - `gets()`, `strcpy()`, `sprintf()`, `scanf()`,...
  - Unsafe programming is often taught in classes and by classical books

What is a BO?

- A *buffer* is a memory space with contiguous chunks of the same data type
  - typically bytes or chars
- We have a *buffer overflow* when a program *writes after the end of a buffer*:
  - or *buffer overrun* in Microsoft jargon
- The problem?
  - C/C++ do not check for these conditions!
  - vs Java/C# that check it in runtime thus solving this problem

What does a BO do?

- What happens when there is an *accidental* BO?
  - Program becomes unstable (*eg, behaves non-deterministically*)
  - Program crashes (*segmentation-fault*)
  - Program proceeds apparently normally
- Side effects depend on:
  - How much data is written after the end of the buffer
  - What data (if any) is overwritten
  - Whether the program tries to read overwritten data
  - What data ends up replacing the memory that gets overwritten
- Debugging a program with such a bug is often hard
  - Effects can appear several execution steps later

Why are BOs a security problem?

- Because they can also be exploited intentionally
- Can let the attacker execute its own code on target machine
  - Objective is usually to run code with superuser privileges
  - …immediate if server running with superuser privileges
  - ...or afterwards using a *privilege escalation attack* to do the rest
  - Reference paper (mainstreamed these attacks): *Aleph One, "Smashing the Stack for Fun and Profit", Phrack 49-14.1996*
- The objective may also be to steal data (next slide)
  - Might be called a *buffer overread* in this case

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

- Arrives in this structure:

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```


Source: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

- Arrives in this structure:

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```


Source: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

- Arrives in this structure:

2 lengths?
1st defined by sender
2nd calculated by recipient

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```


Source: *http://www.theregister.co.uk/2014/04/09/heartbleed_explained/*

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

2 lengths?
1st defined by sender
2nd calculated by recipient

- Arrives in this structure:

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```

- broken OpenSSL code that processes the incoming HeartbeatMessage (p is pointer to the start of the message):

```
/* Read type and payload length first */
hbtype = *p++;      //message type
n2s(p, payload);
     /* copy payload_length into payload
        var and p+=2 */
pl = p;    /*pl now points to payload *(
```


Source: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

- Arrives in this structure:

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```

2 lengths?
1st defined by sender
2nd calculated by recipient

- broken OpenSSL code that processes the incoming HeartbeatMessage (p is pointer to the start of the message):

```
/* Read type and payload length first */
hbtype = *p++;      //message type
n2s(p, payload);
     /* copy payload_length into payload
        var and p+=2 */
pl = p;    /*pl now points to payload *(
```

- Preparing the response:

```
/* Enter response type, len and copy
payload */
*bp++ = TLS1_HB_RESPONSE;  //type
s2n(payload, bp);            //len
memcpy(bp, pl, payload);     //copy
payload
```


Source: *http://www.theregister.co.uk/2014/04/09/heartbleed_explained/*

Read private data with a BO

- Heartbleed vulnerability (April 2014)
- Bug in OpenSSL's implementation of the TLS heartbeat extension (a side sends keep-alive, other replies with same msg)
- Heartbeat message:

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[];
  opaque padding[];
} HeartbeatMessage;
```

- Arrives in this structure:

```
struct ssl3_record_st {
  unsigned int length;
      /* #bytes available */
  [...]
  unsigned char *data;
      /* pointer to Heartbeat message */
  [...]
} SSL3_RECORD;
```

2 lengths?
1st defined by sender
2nd calculated by recipient

- broken OpenSSL code that processes the incoming HeartbeatMessage (p is pointer to the start of the message):

```
/* Read type and payload length first */
hbtype = *p++;      //message type
n2s(p, payload);
     /* copy payload_length into payload
        var and p+=2 */
pl = p;     /*pl now points to payload *(
```

- Preparing the response:

```
/* Enter response type, len and copy
payload */
*bp++ = TLS1_HB_RESPONSE;  //type
s2n(payload, bp);          //len
memcpy(bp, pl, payload);   //copy
payload
```

Attack:

- Small payload (e.g., length = 1)
- Large payload length (e.g., 64K)
- Response returns at most 64KB-1 memory values to requester
- May contain passwords, cripto keys,…

Source: http://www.theregister.co.uk/2014/04/09/heartbleed_explained/

Defending against BOs

- Simple: always do bounds checking
- Problems might arise only when you cannot control input
- C language – Never use **`gets()`**

- Wrong:

```
char buf[1024];
gets (buf);
```

- Right:

```
char buf [BUFSIZE];
fgets (buf, BUFSIZE, stdin);
```

Example: strcpy (I)

- Solution 1

```
if (strlen (src) >= dst_size) {
    /* throw an error */
} else
    strcpy (dst, src)
```

- Solution 2

```
strncpy (dst, src, dst_size - 1);
dst [dst_size - 1] = '\0';
```

- Solution 3:

```
dst = (char *) malloc (strlen(src) + 1);
strcpy (dst, src)
```

Overflowing heap and stack

- Recall segmentation and assume x86
- A program stores data in several places:
  - Global variables – data/bss segments
  - Local variables – stack at the stack segment
  - Dynamic data – heap at the data/bss segments

Stack overflows

Stack smashing (I)

- Stack smashing is the "classical" *stack overflow* attack

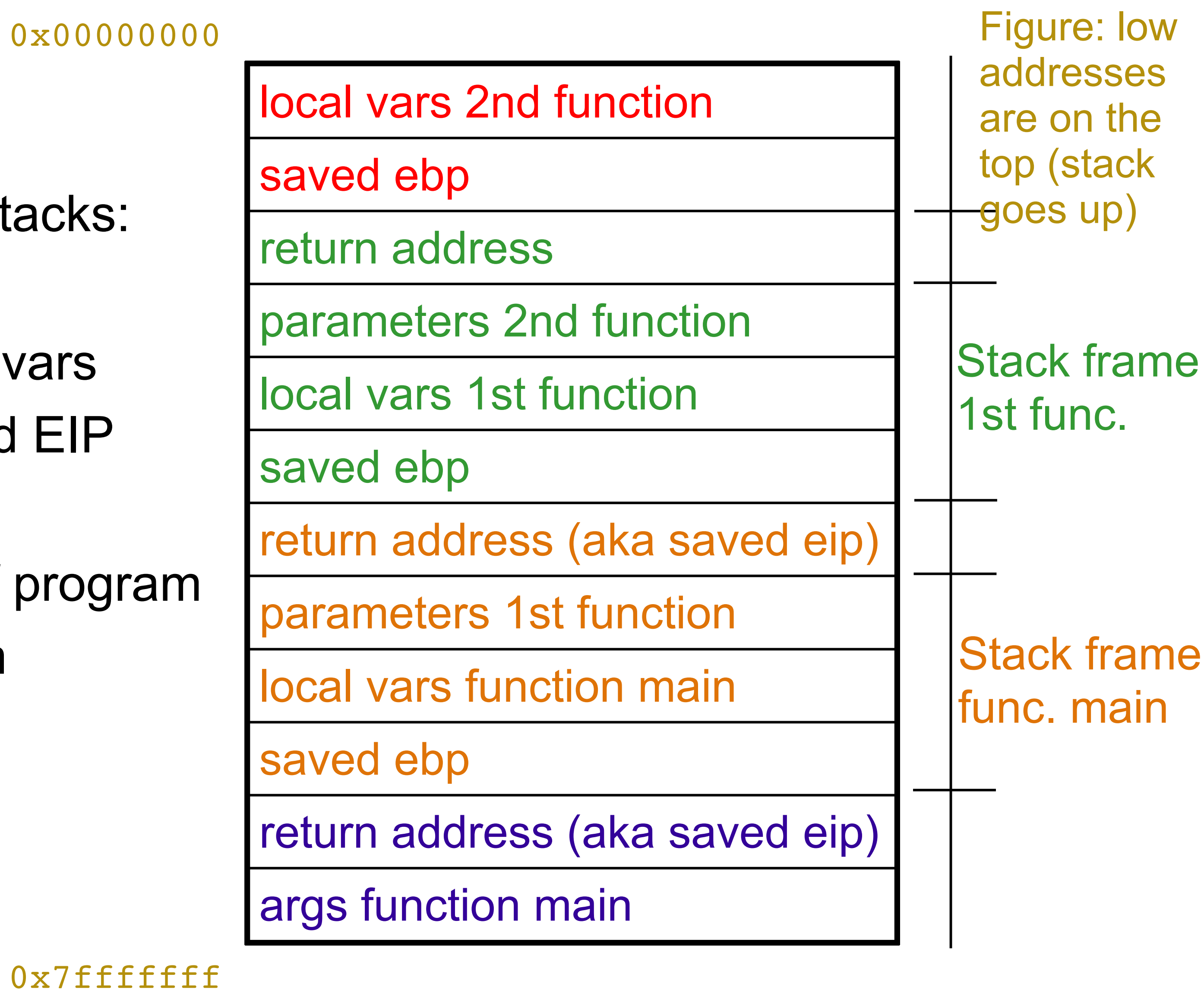
```
void test(char *s) {
  char buf[10];       // gcc stores extra space
  strcpy(buf, s);     // does not check buffer's limit
  printf("&s = %p\n&buf[0] = %p\n\n", &s, buf);
}

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

- *code obviously vulnerable*
  - inserts untrusted input in `buf` without checking
- *gcc* compiles first to assembly...

Stack smashing – stack layout

- StackSmashing attacks:
- Can be:
  - Overflow local vars
  - Overflow saved EIP
- Effects
  - Modify state of program
  - Crash program
  - Execute code



Stack smashing (II)

```
test
   push    ebp
   mov     ebp,esp
   sub     esp,0x14   // buf space
   mov     eax,DWORD PTR [ebp+0x8]
   sub     esp,0x8
   push    eax        // add &s to stack
   lea     eax,[ebp-0x12]
   push    eax        // add &buf to stack
   call    strcpy
   ...
   ret          // jumps to return address
main:
   ...
   call test
```

```
void test(char *s) {

  char buf[10];

  strcpy(buf, s);

  printf("&s = %p\n&buf[0] =
   %p\n\n", s, buf);
}

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

get assembly:
gcc file.c -S

Stack smashing (II)

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14   // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax         // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax         // add &buf to stack
    call    strcpy
    ...
    ret         // jumps to return address
main:
    ...
    call test
```

```
void test(char *s) {

    char buf[10];

    strcpy(buf, s);

    printf("&s = %p\n&buf[0] =
     %p\n\n", s, buf);
}

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

Stack smashing (II)

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14   // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax        // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax        // add &buf to stack
    call    strcpy
    ...
    ret            // jumps to return address
main:
    ...
    call test
```

get assembly:
gcc file.c -S

Stack smashing (II)

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14   // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax        // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax        // add &buf to stack
    call    strcpy
    ...
    ret          // jumps to return address
main:
    ...
    call test
```

```
test:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x14,%esp
    mov    0x8(%ebp),%eax
    sub    $0x8,%esp
    push   %eax
    lea    -0x12(%ebp),%eax
    pushl  %eax
    call   strcpy
    ...
    ret
main:
    …
    call test
```

Stack smashing (II)

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret         // jumps to return address
main:
    ...
    call test
```

Stack

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14      // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax           // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax           // add &buf to stack
    call    strcpy
    ...
    ret           // jumps to return address
main:
    ...
    call test
```

Stack

ret address
arg of test

saved eip

get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret         // jumps to return address
main:
    ...
    call test
```

Stack

saved ebp
ret address
arg of test

saved eip

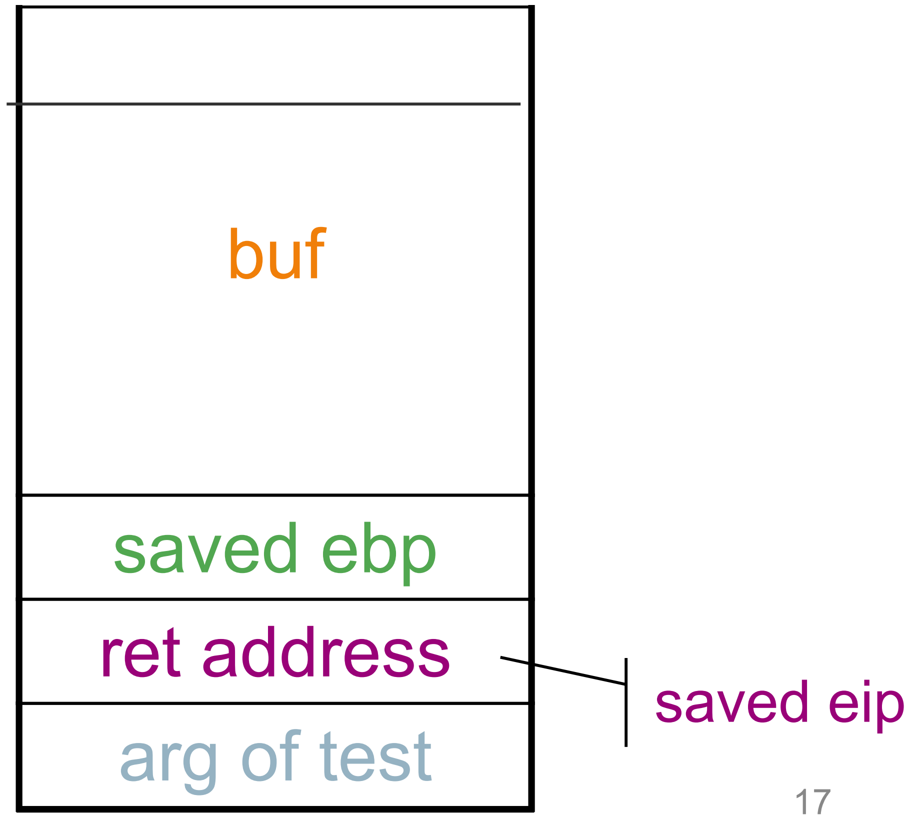
get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret          // jumps to return address
main:
    ...
    call test
```

Stack



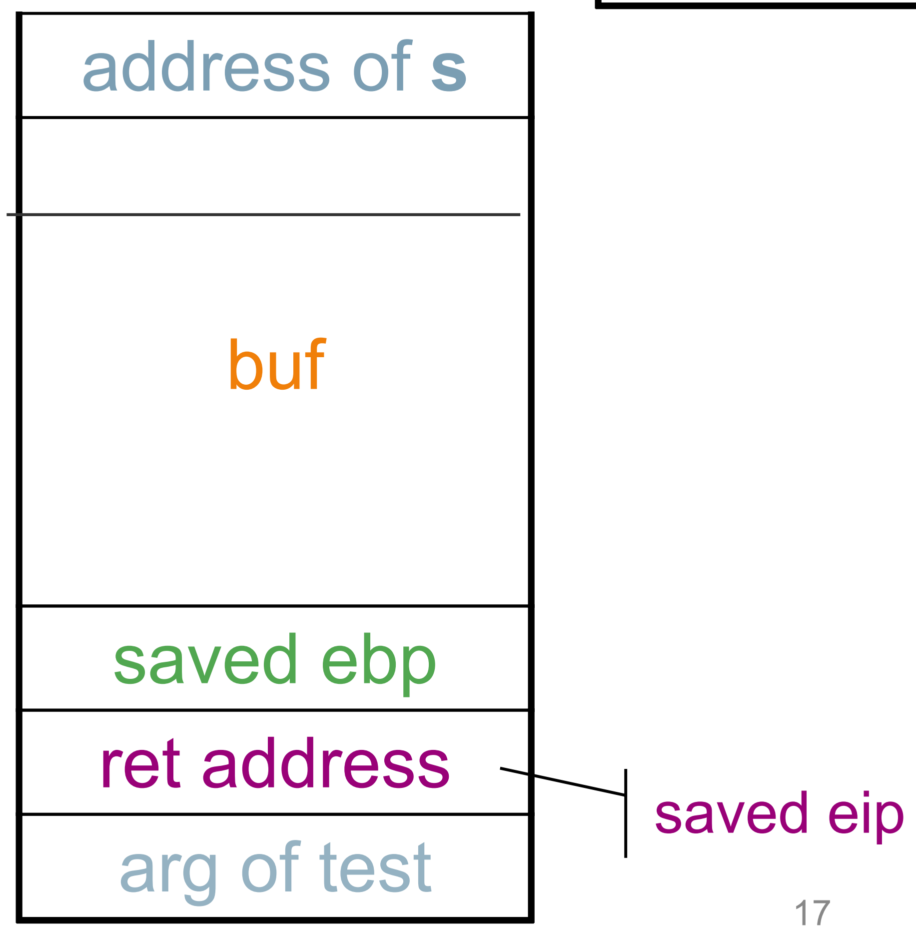
get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret          // jumps to return address
main:
    ...
    call test
```

Stack



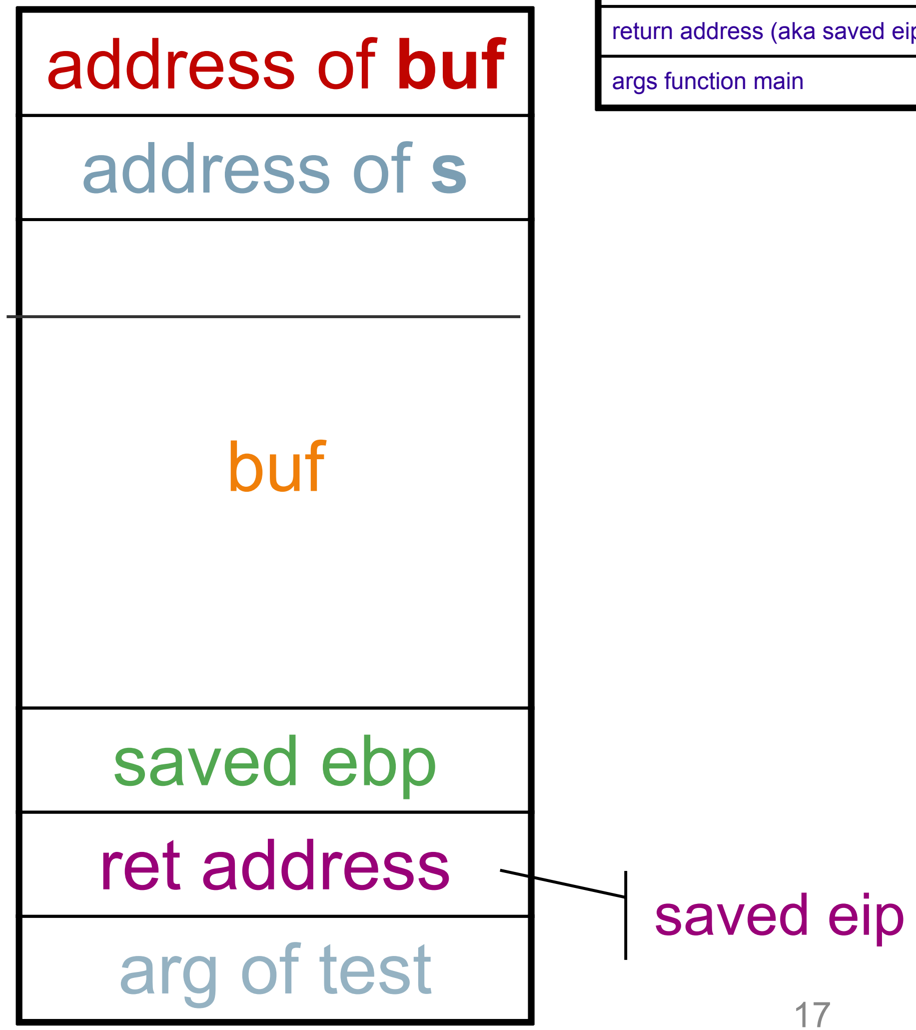
get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret         // jumps to return address
main:
    ...
    call test
```

Stack

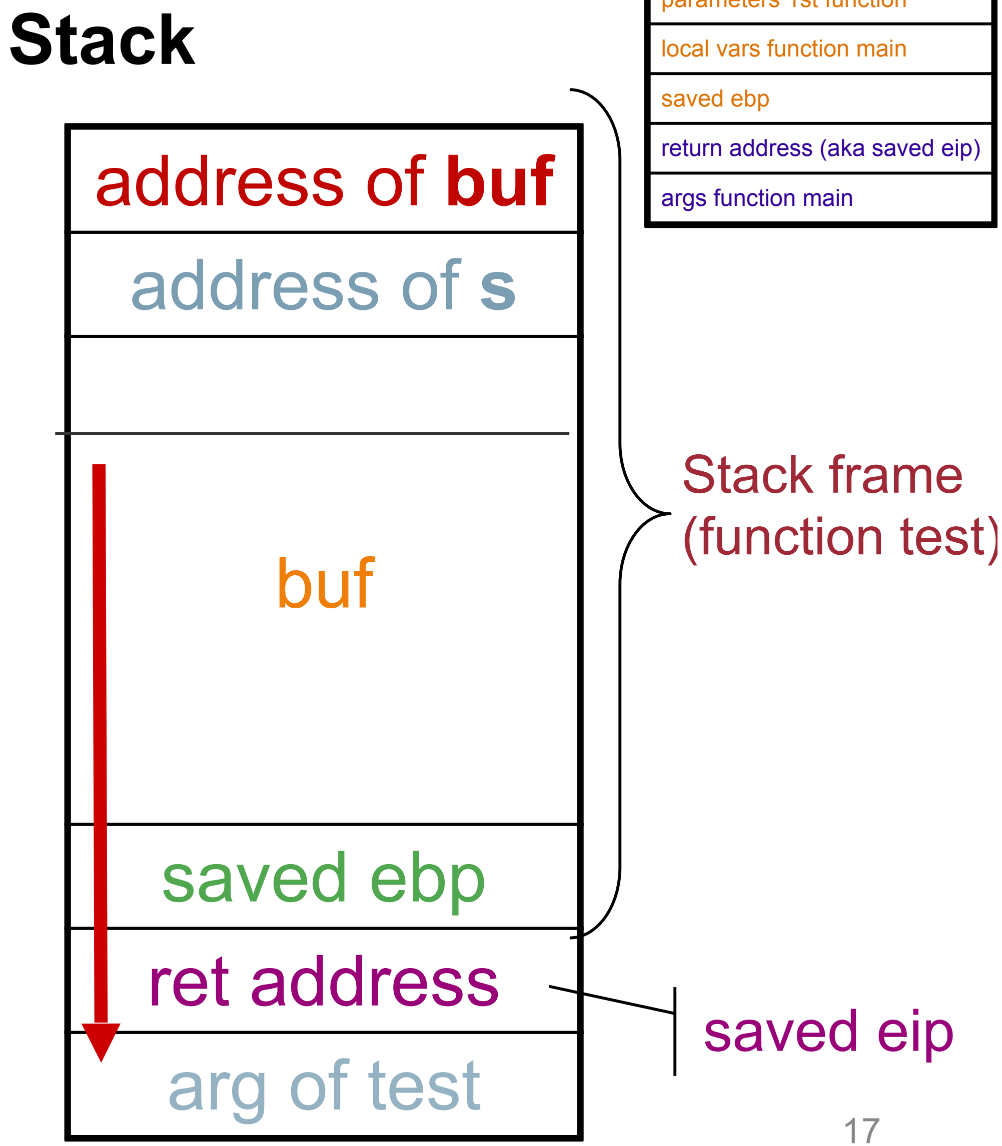


get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret          // jumps to return address
main:
    ...
    call test
```

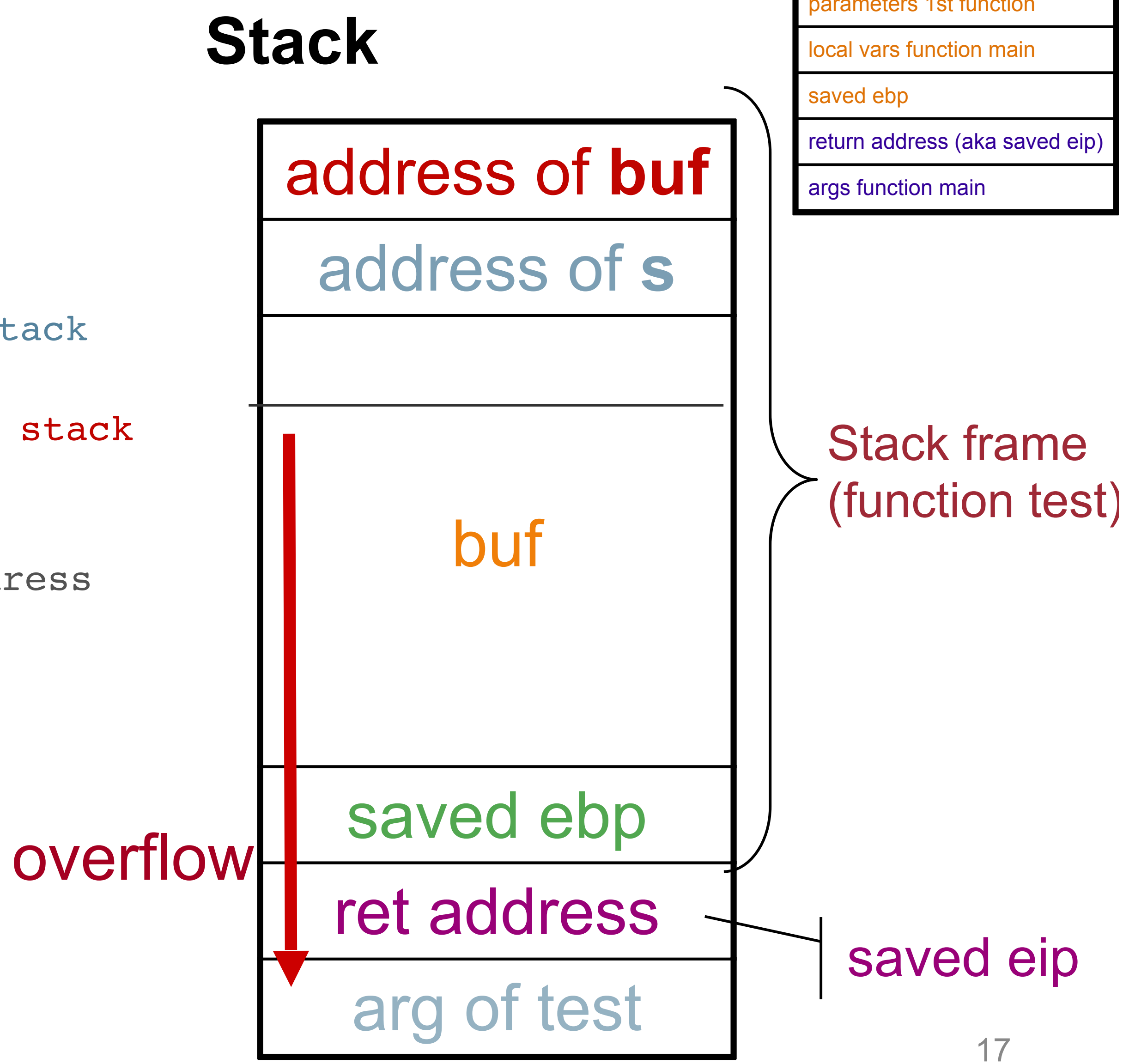


get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14      // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax           // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax           // add &buf to stack
    call    strcpy
    ...
    ret          // jumps to return address
main:
    ...
    call test
```



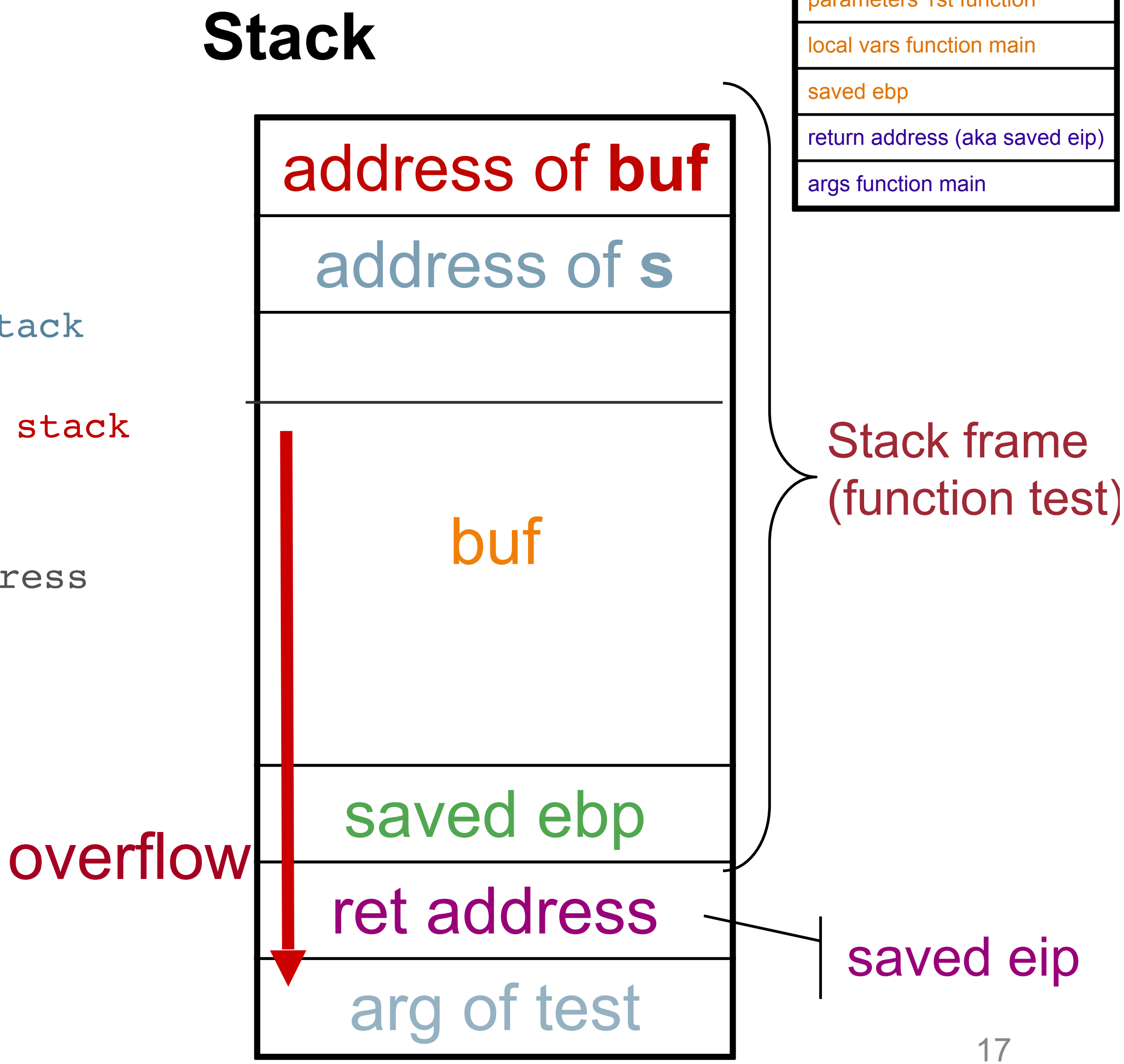
get assembly:
gcc file.c -S

Stack smashing (II)

local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14      // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax           // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax           // add &buf to stack
    call    strcpy
    ...
    ret             // jumps to return address
main:
    ...
    call test
```

Note that local variable **buf** is 0x12 = 18 chars above the `ebp`



get assembly:
gcc file.c -S

Stack smashing (II)

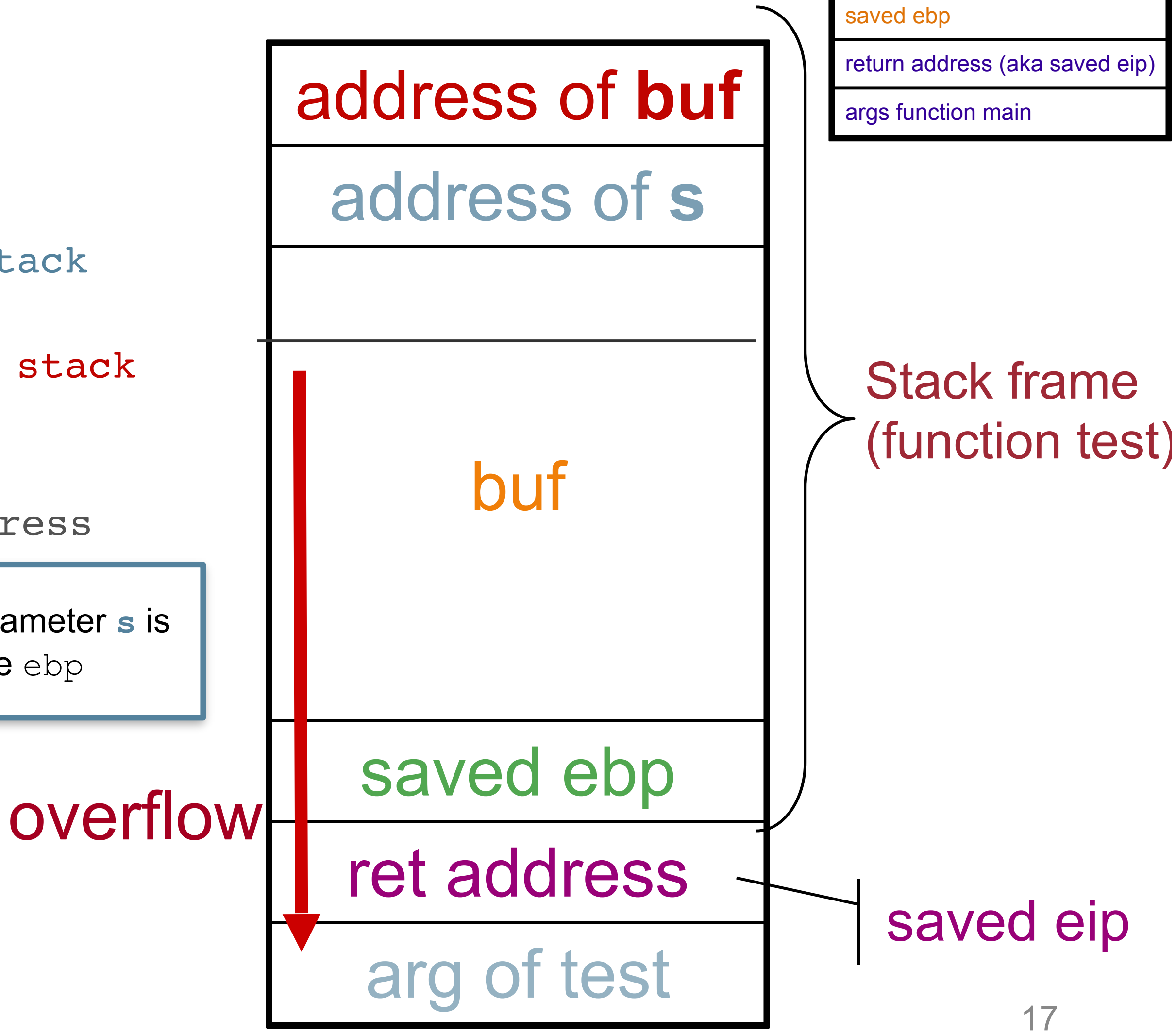
local vars 2nd function
saved ebp
return address
parameters 2nd function
local vars 1st function
saved ebp
return address (aka saved eip)
parameters 1st function
local vars function main
saved ebp
return address (aka saved eip)
args function main

```
test
    push    ebp
    mov     ebp,esp
    sub     esp,0x14     // buf space
    mov     eax,DWORD PTR [ebp+0x8]
    sub     esp,0x8
    push    eax          // add &s to stack
    lea     eax,[ebp-0x12]
    push    eax          // add &buf to stack
    call    strcpy
    ...
    ret           // jumps to return address
main:
    ...
    call test
```

Note that the parameter **s** is 8 chars below the ebp

Note that local variable **buf** is 0x12 = 18 chars above the ebp

Stack



Stack smashing (III)

- Running the previous example:

```
$ ./stack 12345
 &s = 0xffffcea0
 &buf[0] = 0xffffce86

$ ./stack 123456789012345678901234567890
Segmentation fault (core dumped)
```

Stack smashing (IV)

stack_2 – new version of the same program

```
void cannot() {
  puts("This function cannot be executed!\n");
  exit(0);
}

void test(char *s) {
  char buf[10];
  strcpy(buf, s);
  printf(“&s = %p\n&buf[0] = %p\n\n", s, buf);
}

main(int argc, char **argv) {
  printf("&cannot = %p\n", &cannot);
  test(argv[1]);
}
```

?

```
cannot:
    ...
test:
    push    ebp
    mov     ebp,esp
    sub     esp,0x14
    ...
    ret
main:
    ...
    call test
    ...
```

address of **buf**
address of **s**
buf
saved ebp
ret address
arg of test

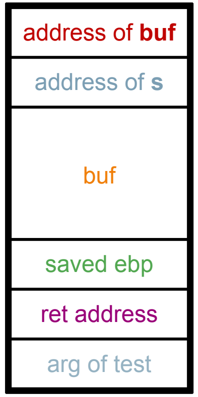
Stack smashing (V)

- Are we able to write the address of `cannot` over the return address to `main`?

```
$ ./stack_2
 &cannot = 0x80484b6
 &s = 0xffffce90
 &buf[0] = 0xffffce76

This function cannot be executed!
```

```
./stack_2 $(python -c 'print("x"*22+"\xb6\x84\x04\x08")')
```



Stack smashing (V)

- Are we able to write the address of `cannot` over the return address to `main`? Consider a program that calls `stack_2`:

```
main() {
  int i;
  char *buf = malloc(1000);
  char **arr = (char **)malloc(10);
  for (i=0; i<22; i++) buf[i] = 'x';
  buf[22] = 0xb6;
  buf[23] = 0x84;
  buf[24] = 0x04;
  buf[25] = 0x08;
  arr[0] = "./stack_2"; arr[1] = buf; arr[2] = 0x00;
  execv("./stack_2", arr);  // executes stack_2 (prev slide)
}
```

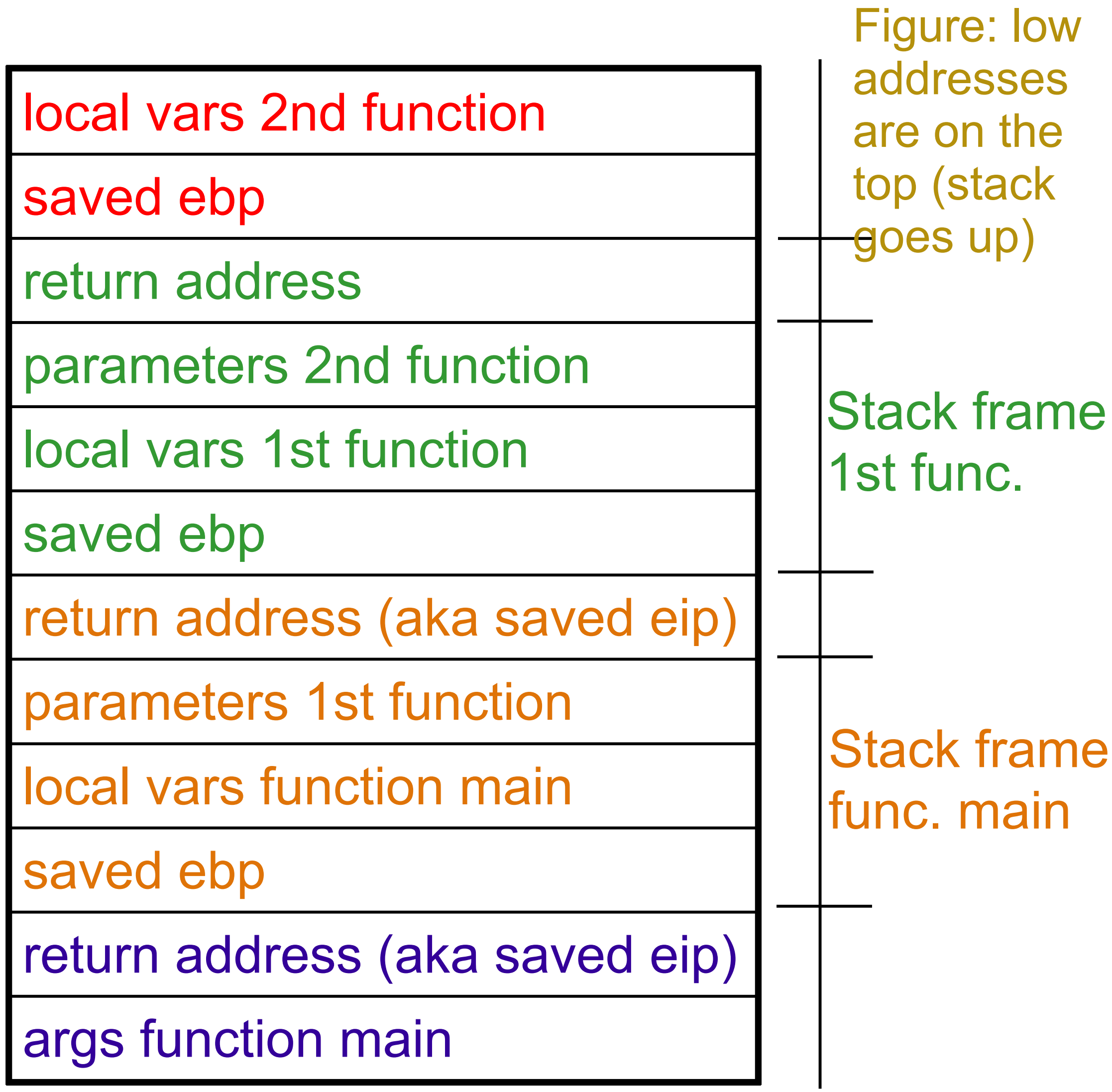
```
$ ./stack_2
 &cannot = 0x80484b6
 &s = 0xffffce90
 &buf[0] = 0xffffce76
```

This function cannot be executed!

Stack smashing – stack layout

- StackSmashing attacks:
- Can be:
  - Overflow local vars
  - Overflow saved EIP
- Effects
  - Modify state of program
  - Crash program
  - Execute code

0x00000000



0x7fffffff

Stack smashing– practical aspects

Stack smashing– practical aspects

- How does the attacker find out the place of the return address?
  - *Without the code:* trial&error (“Smashing the Stack for Fun and Profit”)
  - With the code: analyzing the memory

Stack smashing– practical aspects

- How does the attacker find out the place of the return address?
  - *Without the code:* trial&error ("Smashing the Stack for Fun and Profit")
  - With the code: analyzing the memory
- How does an attacker do something useful with a BO?
  - Inject attack code – **shell code**
  - In Unix, e.g., make program give a shell: `/bin/sh`
  - In Windows, e.g., download rootkit/RAT

Syscall Tables

Show 10 entries Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	**sys_restart_syscall**	0x00	–	–	–	–	–	**kernel/signal.c:2058**
1	**sys_exit**	0x01	int error_code	–	–	–	–	**kernel/exit.c:1046**
2	**sys_fork**	0x02	**struct pt_regs ***	–	–	–	–	**arch/alpha/kernel/entry.S:716**
3	**sys_read**	0x03	unsigned int fd	char __user *buf	size_t count	–	–	**fs/read_write.c:391**
4	**sys_write**	0x04	unsigned int fd	const char __user *buf	size_t count	–	–	**fs/read_write.c:408**
5	**sys_open**	0x05	const char __user *filename	int flags	int mode	–	–	**fs/open.c:900**
6	**sys_close**	0x06	unsigned int fd	–	–	–	–	**fs/open.c:969**
7	**sys_waitpid**	0x07	pid_t pid	int __user *stat_addr	int options	–	–	**kernel/exit.c:1771**
8	**sys_creat**	0x08	const char __user *pathname	int mode	–	–	–	**fs/open.c:933**
9	**sys_link**	0x09	const char __user *oldname	const char __user *newname	–	–	–	**fs/namei.c:2520**

Code injection

- In Unix, the following code can span a shell

```
char *args[] = {"/bin/sh", NULL};
execve("/bin/sh", args, NULL};
```

- in assembly (less than 30 bytes in machine language!):

```
xor %eax, %eax          ;      %eax=0
movl %eax, %edx         ;      %edx = envp = NULL
movl $address_of_path_string, %ebx
movl $address_of_argv, %ecx
movl $0x0b, %al         ; syscall number for
                          execve()
int $0x80               ; do syscall
```

Code injection

- In Unix, the following code can span a shell

```
char *args[] = {"/bin/sh", NULL};
execve("/bin/sh", args, NULL};
```

- in assembly (less than 30 bytes in machine language!):

```
xor %eax, %eax            ;     %eax=0
movl %eax, %edx           ;     %edx = envp = NULL
movl $address_of_path_string, %ebx
movl $address_of_argv, %ecx
movl $0x0b, %al           ; syscall number for
                            execve()
int $0x80                 ; do syscall
```

#	Name	Registers					
		eax	ebx	ecx	edx	esi	edi
11	sys_execve	0x0b	char __user *	char __user * __user *	char __user * __user *	struct pt_regs *	–

Difficulties with code injection

- Lack of space: reduce code
- Code includes zeros/`NULL` bytes (or `\x0D`)
  - (functions like `strcpy` stop in the first zero)
  - Substitute places where zeros appear with equivalent code

    `mov eax, 0` equivalent to `xor eax, eax`
- Discover address where code is injected
  - Some tolerance may be allowed using NOPs in the code
- Stack has to be executable (usually is)
  - But if it doesn't there are other forms of injection --- next:

Arc injection *or* return-to-libc

- Insert a new ***arc*** in the program control-flow *graph*
  - (stack-smashing includes a new *node* in the graph)
  - e.g., overrun the return address to point to code already in the program – typically to the **`system`** function of the *libc* (return to libc)

Arc injection *or* return-to-libc

- Insert a new ***arc*** in the program control-flow *graph*
  - (stack-smashing includes a new *node* in the graph)
  - e.g., overrun the return address to point to code already in the program – typically to the **`system`** function of the *libc* (return to libc)
- Attack against the *system* function:
  - Register *R* has to contain the address of attacker supplied *data*
    - But registers are reused and often point to buffer in the stack
  - *Return address* is set to *target*, causing the processor to jump there

```
void system(char *arg){
     check_validity(arg); //bypass this
     R=arg;
 target: execl(R, …);  //target is usually fixed
```

Arc injection *or* return-to-libc

- Insert a new ***arc*** in the program control-flow *graph*
  - (stack-smashing includes a new *node* in the graph)
  - e.g., overrun the return address to point to code already in the program – typically to the **`system`** function of the *libc* (return to libc)
- Attack against the *system* function:
  - Register *R* has to contain the address of attacker supplied <u>*data*</u>
    - But registers are reused and often point to buffer in the stack
  - *Return address* is set to *target*, causing the processor to jump there

```
void system(char *arg){
     check_validity(arg); //bypass this
     R=arg;
 target: execl(R, …);  //target is usually fixed
```

Stack does not have to be executable

Pointer subterfuge

- General term for exploits that involve *modifying a pointer*
  - Objective can be to circumvent protections against BOs – return address protected
- *Four varieties:*

1. Function-pointer clobbering
   - Modify a function pointer to point to attacker supplied code
2. Data-pointer modification
   - Modify address used to assign data
3. Exception-handler hijacking
   - Modify pointer to an exception handler function
4. Virtual pointer smashing
   - Modify the C++ virtual function table associated with a class

buff
f
saved ebp
ret address
arg of test

1. Function-pointer clobbering

- Modify function pointer to point to the code desired by the attacker (e.g. supplied by him)

```
void f2a(void * arg, size_t len) {
   void (*f)() = ...;           /* function pointer */
   char buff[100];
   memcpy(buff, arg, len);      /* buffer overflow! */
   f();
   /* ... */
   return;
}
```

buff
f
saved ebp
ret address
arg of test

1. Function-pointer clobbering

- Modify function pointer to point to the code desired by the attacker (e.g. supplied by him)

```
void f2a(void * arg, size_t len) {
   void (*f)() = ...;          /* function pointer */
   char buff[100];
   memcpy(buff, arg, len);     /* buffer overflow! */
   f();
   /* ... */
   return;
}
```

Overwrite **f** with address of malicious code in **buff**

Call function **f**…

buff
f
saved ebp
ret address
arg of test

1. Function-pointer clobbering

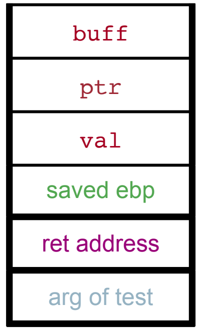
- Modify function pointer to point to the code desired by the attacker (e.g. supplied by him)

```
void f2a(void * arg, size_t len) {
   void (*f)() = ...;          /* function pointer */
   char buff[100];
   memcpy(buff, arg, len);     /* buffer overflow! */
   f();
   /* ... */
   return;
}
```

Overwrite **`f`** with address of malicious code in **`buff`**

Call function **`f`**…

Combines well with *arc injection* (e.g., overflow **`f`** with pointer to **`system`**).



2. Data-pointer modification

- A pointer used to assign a value is controlled by an attacker for <u>*arbitrary memory write*</u>

```
void f2b(void * arg, size_t len) {
    long val = ...;
    long *ptr = ...;
    char buff[100];
    extern void (*f)();
    memcpy(buff, arg, len);  /* buffer overflow! */
    *ptr = val;
    f();
    /* ... */
    return; }
```

buff
ptr
val
saved ebp
ret address
arg of test

2. Data-pointer modification

- A pointer used to assign a value is controlled by an attacker for *arbitrary memory write*

```
void f2b(void * arg, size_t len) {
    long val = ...;
    long *ptr = ...;
    char buff[100];
    extern void (*f)();
    memcpy(buff, arg, len);  /* buffer overflow! */
    *ptr = val;
    f();
    /* ... */
    return; }
```

Function pointer **f** is not local, thus not prone to function pointer clobbering. But with data-pointer modification, **f** can be overwritten…

2. Data-pointer modification

buff
ptr
val
saved ebp
ret address
arg of test

- A pointer used to assign a value is controlled by an attacker for ***arbitrary memory write***

```
void f2b(void * arg, size_t len) {
    long val = ...;
    long *ptr = ...;
    char buff[100];
    extern void (*f)();
    memcpy(buff, arg, len);  /* buffer overflow! */
    *ptr = val;
    f();
    /* ... */
    return; }
```

Function pointer **`f`** is not local, thus not prone to function pointer clobbering. But with data-pointer modification, **`f`** can be overwritten…

A BO can overwrite **`ptr`** and **`val`**, allowing to write 4 bytes arbitrarily in the memory

3. Exception-handler hijacking

- Windows Structured Exception Handler (SEH)
  - When an exception is generated, Windows examines a linked list of exception handlers descriptors, then invokes the corresponding handler (function pointer)
  - The list is in the stack, so it can be overrun
- Attack:
  - The addresses of the handlers are substituted by pointers to attacker supplied code or other places (e.g. libc)
  - A exception is caused in some way (e.g., writing over all the stack causes an exception when its base is overwritten)
- Validity checking of the SEH is done since Windows Server 2003

4. Virtual pointer smashing

- Most C++ compilers use a *virtual function table (VTBL)* associated with each class
  - Array of function pointers
  - An object has in its header a *virtual pointer (VPTR)* to its class VTBL
  - An attacker might overrun the VPTR of an object with a pointer to a fake VTBL (with pointers to attacker supplied code, libc,…)

```
void f4(void * arg, size_t len) {
    C *ptr = new C;
    char *buff = new char[100];
    memcpy(buff, arg, len); /* buffer overflow! */
    ptr->vf();          // call to a virtual function
    return; }
```

Off-by-one errors

- Is this code correct?

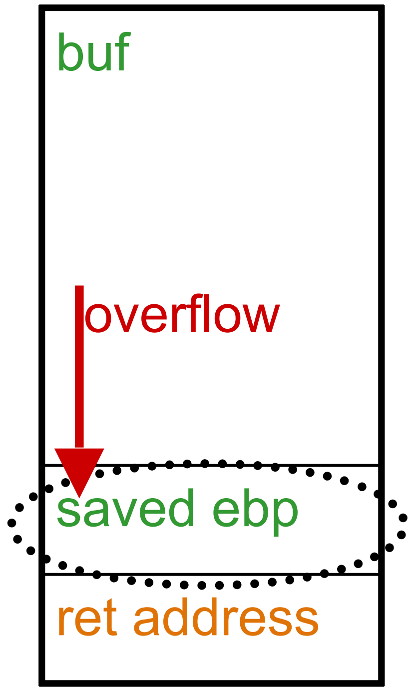
```
int get_user(char *user) {
    char buf[1024];
    if (strlen(user) > sizeof(buf))
        handle_error (“string too long”);
    strcpy(buf, user);
}
```

Off-by-one errors

- Is this code correct?

```
int get_user(char *user) {
    char buf[1024];
    if (strlen(user) > sizeof(buf))
        handle_error ("string too long");
    strcpy(buf, user);
}
```

- BO of 1 char (`\0`) if *sizeof(user)=1024*
  - Is this exploitable? Only 1 byte…
  - Saved ebp has 4 bytes, 80x86 is little endian, so LSB is put to 0 ➔ saved ebp is reduced by 0 to 255 bytes
  - ➔ It can be as if the next frame is in the buffer! Local variables or the return address can be modified…

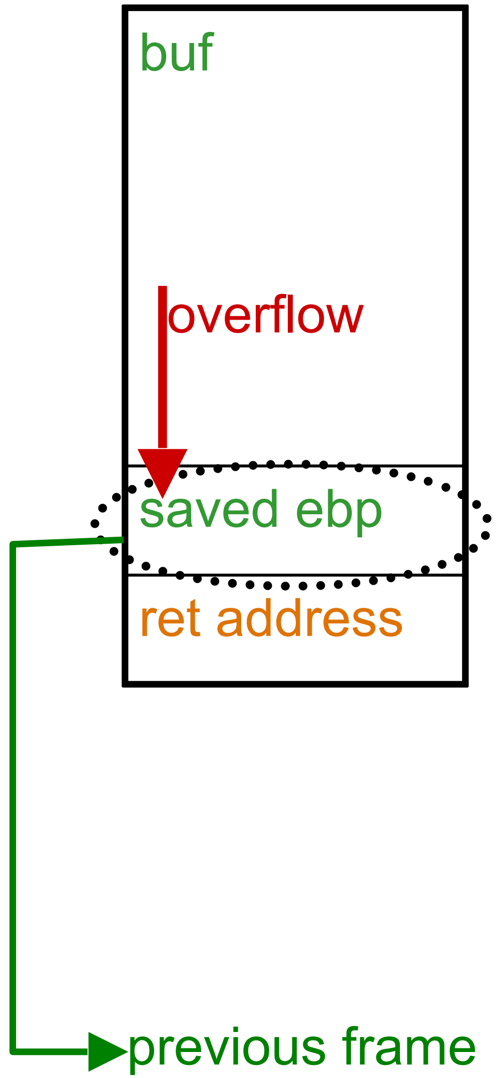


Off-by-one errors

- Is this code correct?

```
int get_user(char *user) {
    char buf[1024];
    if (strlen(user) > sizeof(buf))
        handle_error ("string too long");
    strcpy(buf, user);
}
```

- BO of 1 char (`\0`) if *sizeof(user)=1024*
  - Is this exploitable? Only 1 byte…
  - Saved ebp has 4 bytes, 80x86 is little endian, so LSB is put to 0 ➔ saved ebp is reduced by 0 to 255 bytes
  - ➔ It can be as if the next frame is in the buffer! Local variables or the return address can be modified…

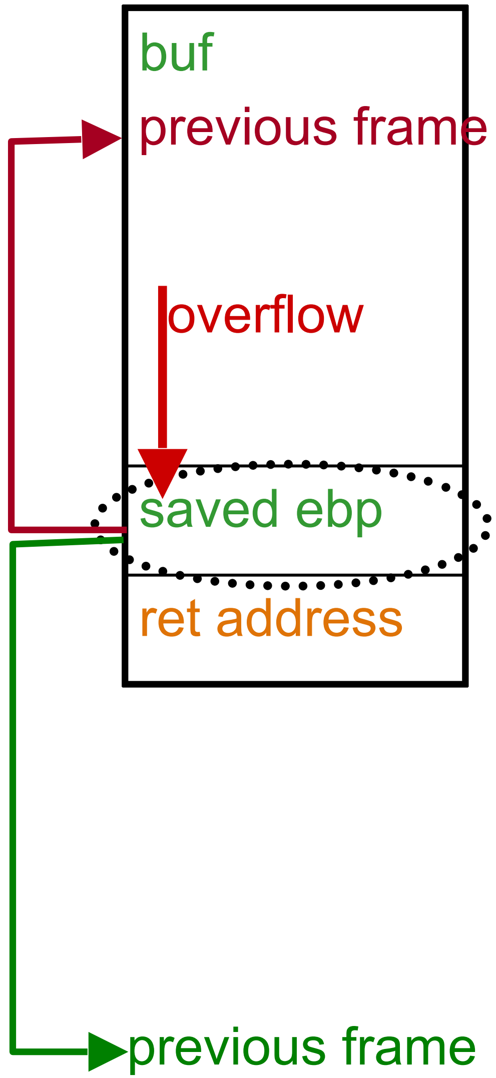


Off-by-one errors

- Is this code correct?

```
int get_user(char *user) {
    char buf[1024];
    if (strlen(user) > sizeof(buf))
        handle_error ("string too long");
    strcpy(buf, user);
}
```

- BO of 1 char (`\0`) if *sizeof(user)=1024*
  - Is this exploitable? Only 1 byte…
  - Saved ebp has 4 bytes, 80x86 is little endian, so LSB is put to 0 ➔ saved ebp is reduced by 0 to 255 bytes
  - ➔ It can be as if the next frame is in the buffer! Local variables or the return address can be modified…



Return-Oriented Programming attacks

- Motivation:
  - return-to-libc doesn't work well in 64-bit CPUs due to convention that 1st function parameters put in registers
- ROP provides a "language"
  - attacker can use it to make a machine do something useful
  - it involves no code injection; generalizes return-to-libc
- gadget: sequence of instructions ending with `ret` (C3)
  - x86 machine language is dense: most short sequences of bytes are valid instructions
  - finding gadgets: search for C3 and see what instructions can be found before it

ROP attacks (2)

- Unintended instructions
  - Gadgets aren't necessarily based on instructions of the original code
  - Example of piece of code with 3 instructions (no ret) that becomes a ga

Original instruction	Byte	Unintended instruction
movl $0x00000001, -44(%ebp)	c7	
	45	
	d4	
	01	
	00	
	00	
	00	add %dh, %bh
test $0x00000007, %edi	f7	
	c7	movl $0x0F000000, (%edi)
	07	
	00	
	00	
	00	
setnzb -61(%ebp)	0f	
	95	xchg %ebp, %eax
	45	inc%ebp
	c3	ret

ROP attacks (3)

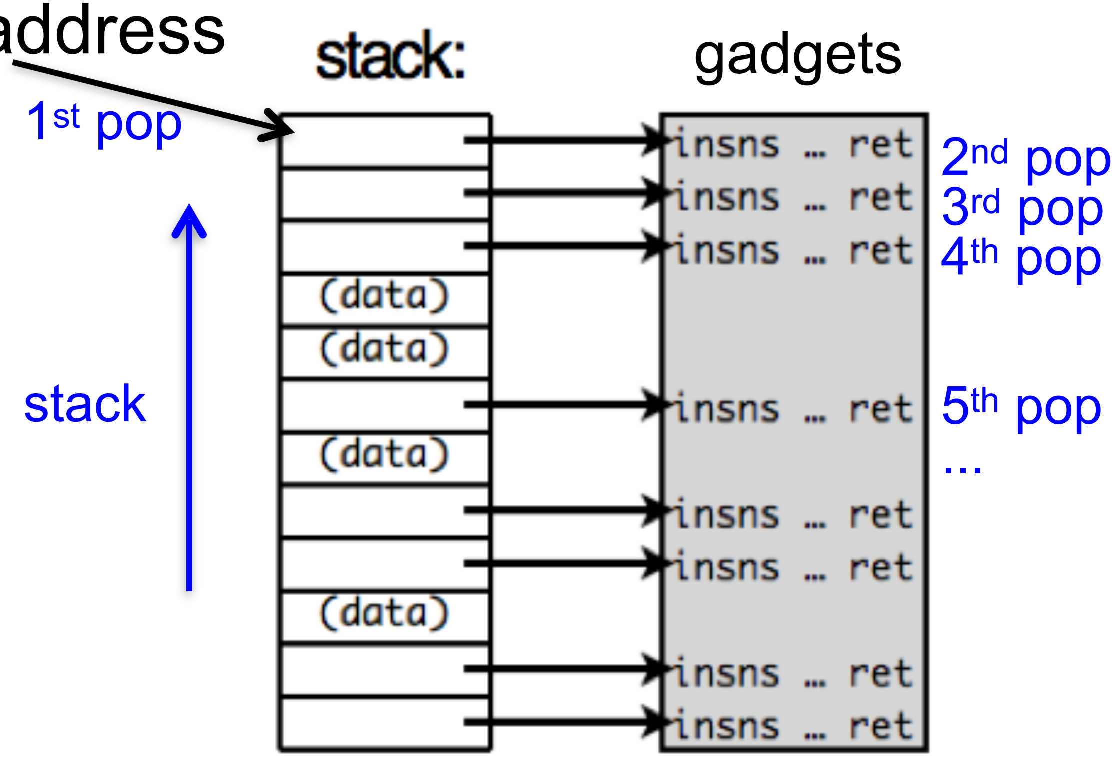
- Finding gadgets (instruction sequences ending in ret)

```
for every instance of C3 in the binary code
    call search_for_instruction(position)
function search_for_instruction(position) //for 1 inst before position
    for n=1 to …
        if the n bytes before position are a valid n-byte instruction
            we have an instruction
            call search_for_instruction(position-n)
```

- At the end, we have a large set of gadgets
  – Some are trash, e.g., `pop %ebp; ret`

ROP attacks (4)

- Running the attack is to overflow the stack with
  - (1) addresses of gadgets
  - (2) other data the gadgets may pick from the stack
  - Address of 1st gadget to call must be on top of the return address



Integer Overflows

Integer overflows – basics

- The semantics of integer-handling is complex and programmers often don't know the details
  - Problem in several languages, but especially in C / C++
  - E.g., what happens when a signed integer is passed to a unsigned?
  - 4 problems: overflow, underflow, signedness error, truncation
  - First two also in *type safe languages* (Java, C#)
- 5 exploits:
  - Insufficient memory allocation → BO → attacker code execution
  - Excessive memory allocation / infinite loop → denial of service
  - Attack against array byte index → overwrite arbitrary byte in mem.
  - Attack to bypass sanitization → cause a BO → …
  - Logic errors (e.g. modify variable to modify program behavior)
- We consider the standard C99, integer representation convention ILP32
  - int=long=pointer=32bits, integers represented in 2's complement

1. Overflow

- Result of expression exceeds maximum value of the type
- The most common integer overflow form
  - 148 out of 207 CVE vulnerabilities in Brumley et al.'s study
- Ex. from GOCR OCR program
  - If `x*y>MAXINT`, malloc does not reserve enough mem

```
void vulnerable(char *matrix,
 size_t x, size_t y, char val){
 int i, j;
 matrix = (char *) malloc (x*y);
 for (i=0; i<x; i++){
      for (j=0; j<y; j++){
            matrix[i*y+j] = val;
      }
 }
}
```

size_t is the same as unsigned int

2. Underflow

size_t = unsigned

- Result of expression is smaller than the minimum value of the type
  - E.g., subtracting 0-1 and storing the result in an unsigned int
  - Rarer as only with subtraction, not with other operations
  - Netscape JPEG comment length vulnerability:

```
void vulnerable(char *src, size_t len){
  size_t real_len;
  char *dst;
  if (len < MAX_SIZE) {
    real_len = len - 1;
    dst = (char *) malloc (real_len);
    memcpy(dst, src, real_len);
   }
}
```

2. Underflow

size_t = unsigned

- Result of expression is smaller than the minimum value of the type
  - E.g., subtracting 0-1 and storing the result in an unsigned int
  - Rarer as only with subtraction, not with other operations
  - Netscape JPEG comment length vulnerability:

```
void vulnerable(char *src, size_t len){
  size_t real_len;
  char *dst;
  if (len < MAX_SIZE) {
    real_len = len - 1;
    dst = (char *) malloc (real_len);
    memcpy(dst, src, real_len);
    }
}
```

If `len`=0, then `real_len`= FFFFFFFF, and malloc allocs FFFFFFFF bytes

3. Signedness error

size_t = unsigned

- A signed integer is interpreted as unsigned or vice-versa
  - Negative number interpreted as positive → the sign bit (1) is interpreted as 2^{31}
  - 44 out of the 195 CVE vulnerabilities in Brumley et al.'s study
  - Linux kernel XDR vulnerability:

```
1 void vuln(char *src, size_t len){
2  int real_len;
3  char *dst;
4  if (len > 1) {
5    real_len = len - 1;
6    if (real_len < MAX_SIZE) {
7      dst=(char *) malloc(real_len);
8      memcpy(dst, src, real_len);
9      }
10    }
11 }
```

Signed integers are represented in 2's complement in ILP32

Line 5: negative if `len` > 2^{31}

4. Truncation

- Assigning an integer with a longer width to another shorter
  - Ex: assigning an int (32 bits) to a short (16 bits)
  - SSH CRC-32 compensation attack detector vulnerability:

```c
void vuln(char *src, unsigned len){
  unsigned short real_len;
  char *dst;
  real_len = len;
  if (real_len < MAX_SIZE) {
    dst = (char *) malloc(real_len);
    strcpy(dst, src);
    }
}
```

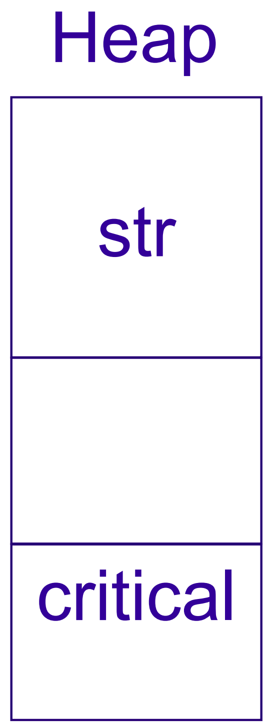
A large packet causes a truncation → *malloc* allocs too little space → the code that initializes the space corrupts the memory

Heap overflows

Heap overflow – basic (I)

- Modify value of data in the heap

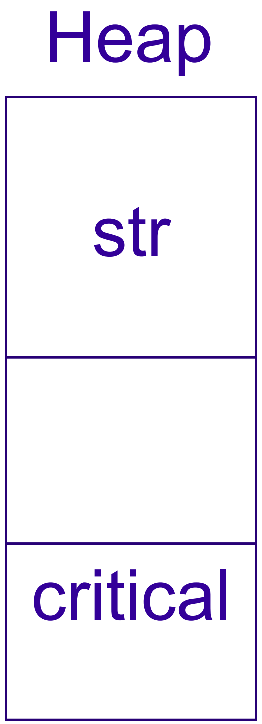
```
main(int argc, char **argv) {
   int i;
   char *str = (char *)malloc(4);
   char *critical = (char *)malloc(9);
   strcpy(critical, "secret");
   strcpy(str, argv[1]);
   printf("%s\n", critical);
}
```



Heap overflow – basic (I)

- Modify value of data in the heap

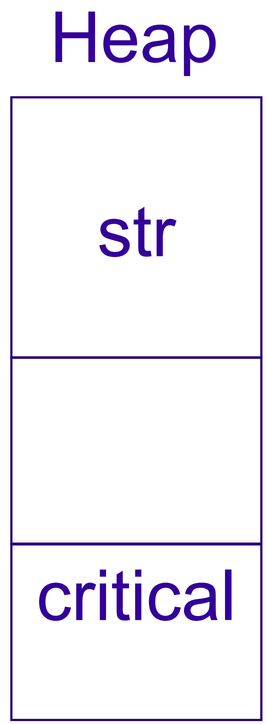
```
main(int argc, char **argv) {
  int i;
  char *str = (char *)malloc(4);
  char *critical = (char *)malloc(9);
  strcpy(critical, "secret");
  strcpy(str, argv[1]);
  printf("%s\n", critical);
}
```



Heap overflow – basic (I)

- Modify value of data in the heap

```
main(int argc, char **argv) {
   int i;
   char *str = (char *)malloc(4);
   char *critical = (char *)malloc(9);
   strcpy(critical, "secret");
   strcpy(str, argv[1]);
   printf("%s\n", critical);
}
```



Heap overflow – basic (II)

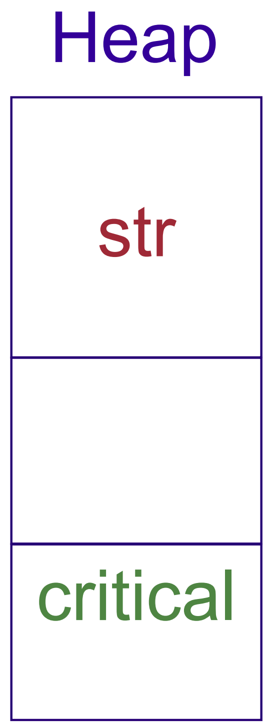
```
main(int argc, char **argv) {
  int i;
  char *str = (char *)malloc(4);
  char *critical = (char *)malloc(9);
  char *tmp;
  printf("Address of str is: %p\n", str);
  printf("Address of critical is: %p\n", critical);
  strcpy(critical, "secret");
  strcpy(str, argv[1]);
  tmp = str;
  while(tmp < critical+9) {     // print heap content
    printf("%p: %c (0x%x)\n",
        tmp, isprint(*tmp) ? *tmp: '?', (unsigned)(*tmp));
    tmp += 1;
  }
  printf("%s\n", critical);
}
```

Heap overflow – basic (III)

./a.out xyz

Address of str is: 0x80497e0
Address of critical is: 0x80497f0
0x80497e0: x (0x78)
0x80497e1: y (0x79)
0x80497e2: z (0x7a)
0x80497e3: ? (0x0)
0x80497e4: ? (0x0)
0x80497e5: ? (0x0)
0x80497e6: ? (0x0)
0x80497e7: ? (0x0)
0x80497e8: ? (0x0)
0x80497e9: ? (0x0)
0x80497ea: ? (0x0)
0x80497eb: ? (0x0)
0x80497ec: ? (0x11)
0x80497ed: ? (0x0)
0x80497ee: ? (0x0)
0x80497ef: ? (0x0)
0x80497f0: s (0x73)
0x80497f1: e (0x65)
0x80497f2: c (0x63)
0x80497f3: r (0x72)
0x80497f4: e (0x65)
0x80497f5: t (0x74)
0x80497f6: ? (0x0)
0x80497f7: ? (0x0)
0x80497f8: ? (0x0)

secret

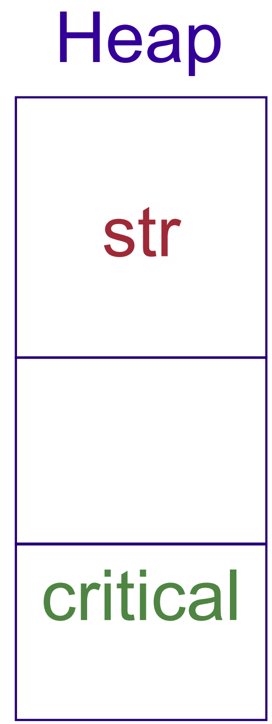


Heap overflow – basic (IV)

./a.out xyz1234567890123COVFEFE

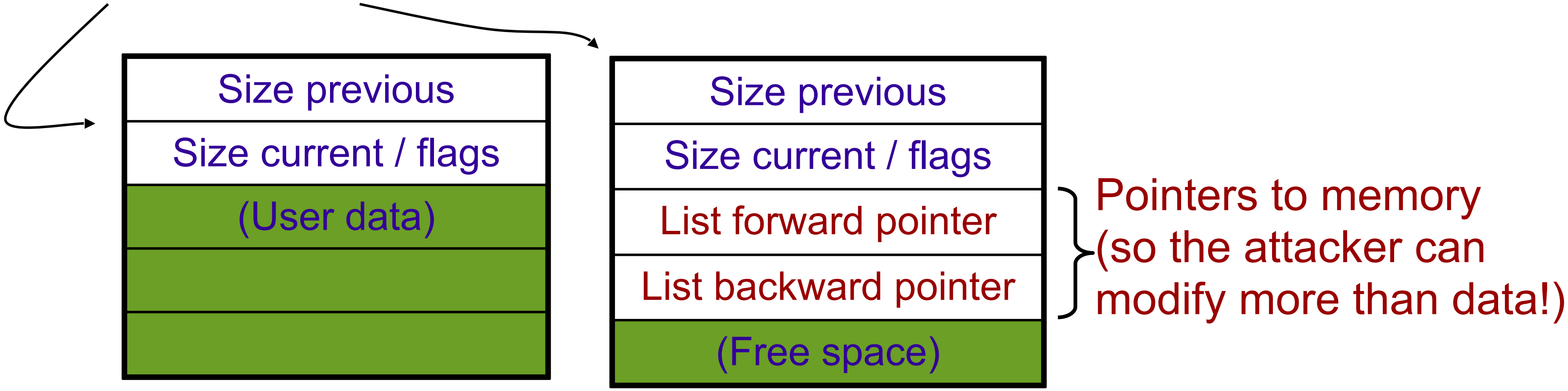
Address of str is: 0x80497f0
Address of critical is: 0x8049800
0x80497f0: x (0x78)
0x80497f1: y (0x79)
0x80497f2: z (0x7a)
0x80497f3: 1 (0x31)
0x80497f4: 2 (0x32)
0x80497f5: 3 (0x33)
0x80497f6: 4 (0x34)
0x80497f7: 5 (0x35)
0x80497f8: 6 (0x36)
0x80497f9: 7 (0x37)
0x80497fa: 8 (0x38)
0x80497fb: 9 (0x39)
0x80497fc: 0 (0x30)
0x80497fd: 1 (0x31)
0x80497fe: 2 (0x32)
0x80497ff: 3 (0x33)
0x8049800: C (0x43)
0x8049801: O (0x4F)
0x8049802: V (0x56)
0x8049803: F (0x46)
0x8049804: E (0x45)
0x8049805: F (0x46)
0x8049806: E (0x45)
0x8049807: ? (0x0)
0x8049808: ? (0x0)

COVFEFE



Heap overflow – advanced (I)

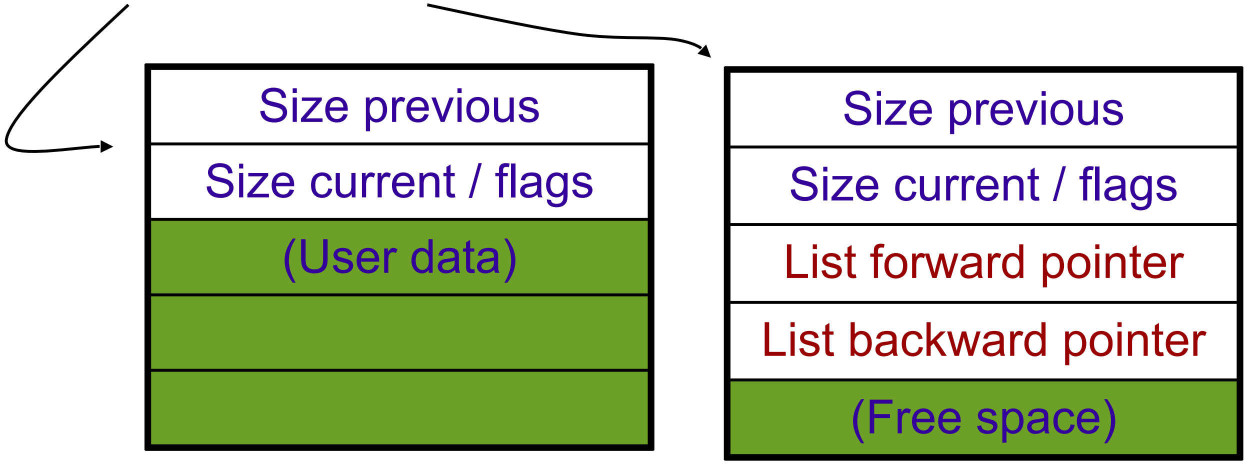
- Problem (for the hacker): heap implementations vary much
  - malloc: gets a block of data
  - free: frees a block (typically only marks it “free”)
- Free blocks usually chained using a double-linked list
- Usually blocks are stored with control data in-band:
  - size, link to next free, free/in-use flag, etc.
  - In-use and free blocks in **glibc** (GNU’s libc):



Heap overflow – advanced (I)

- Problem (for the hacker): heap implementations vary much
  - malloc: gets a block of data
  - free: frees a block (typically only marks it “free”)
- Free blocks usually chained using a double-linked list
- Usually blocks are stored with control data in-band:
  - size, link to next free, free/in-use flag, etc.
  - In-use and free blocks in **glibc** (GNU’s libc):

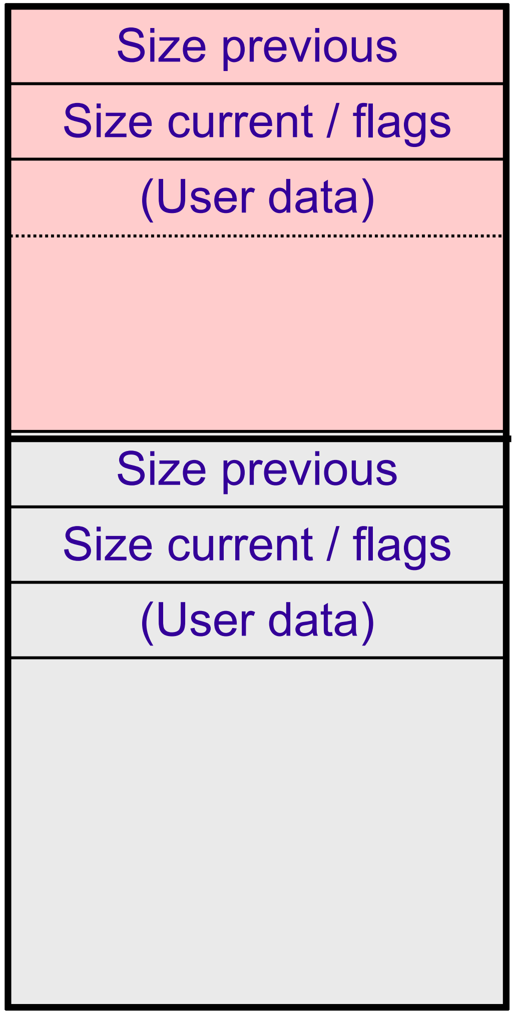
BSD: control data is stored out-off-band precisely to prevent these attacks



Pointers to memory (so the attacker can modify more than data!)

Heap overflow – advanced (II) …

Start with 2 blocks
(both occupied)



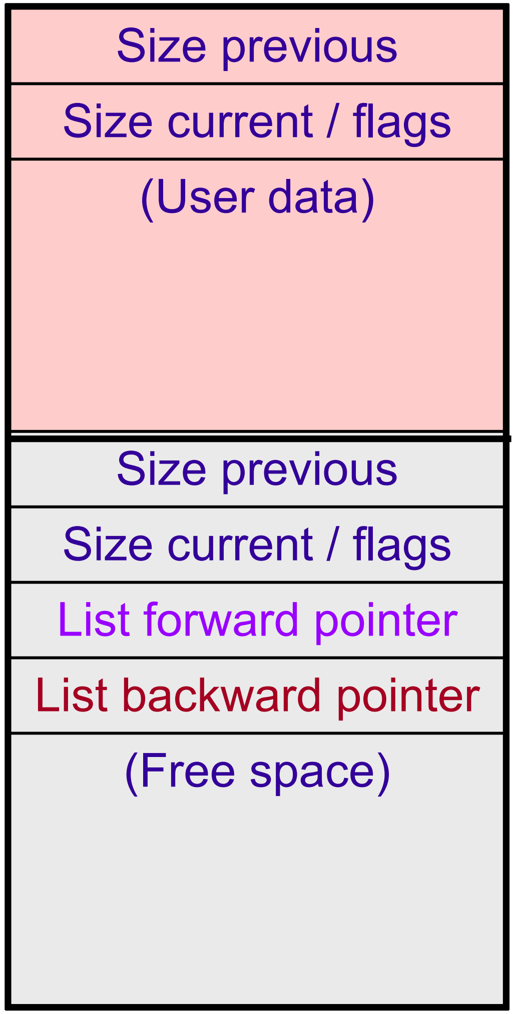
…

Heap overflow – advanced (II)

Start with 2 blocks
(both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers

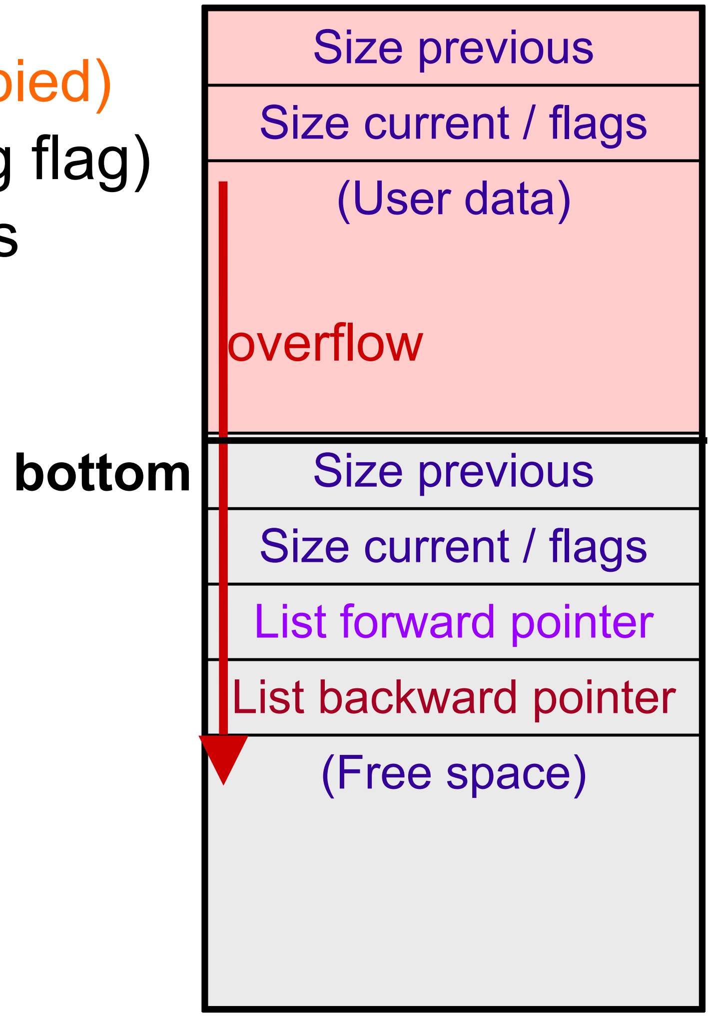
bottom



Heap overflow – advanced (II)

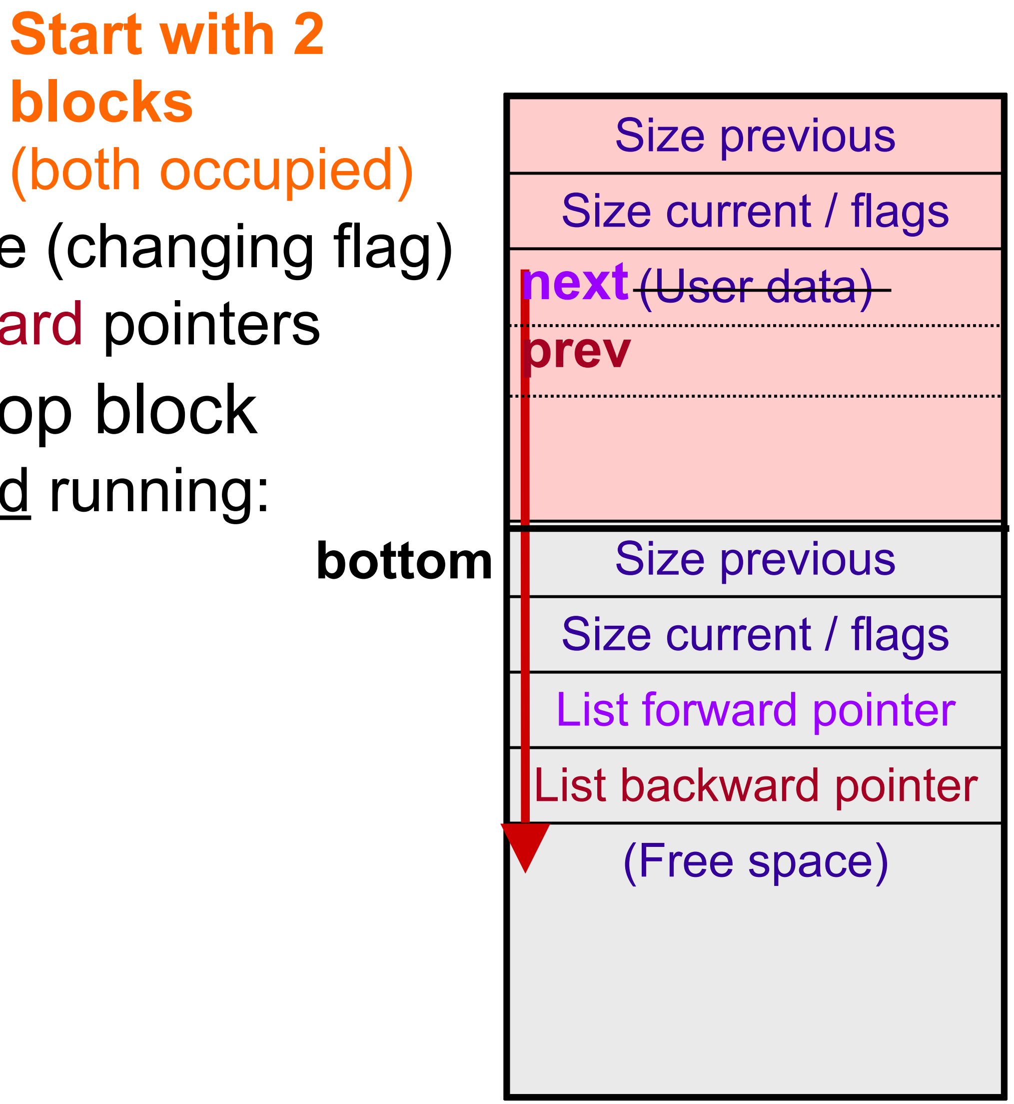
Start with 2 blocks (both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers



Heap overflow – advanced (II) …

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers
- 2nd step – program frees top block
  - The two blocks are merged running:



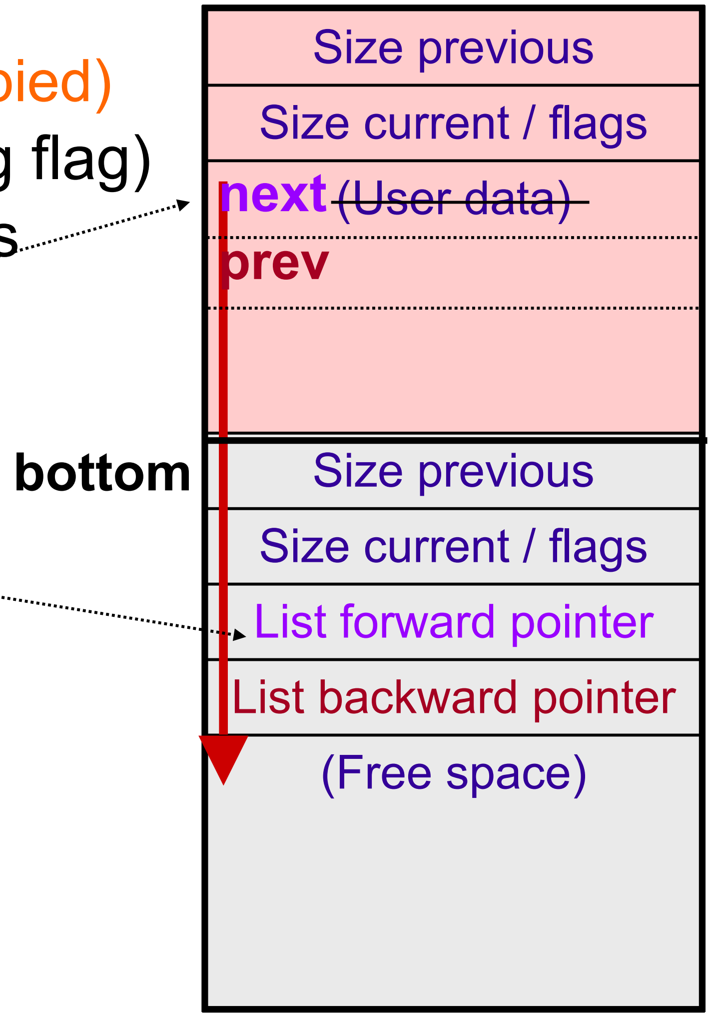
…

Heap overflow – advanced (II) …

Start with 2 blocks (both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers
- 2nd step – program frees top block
  - The two blocks are merged running:

    next= bottom->next



…

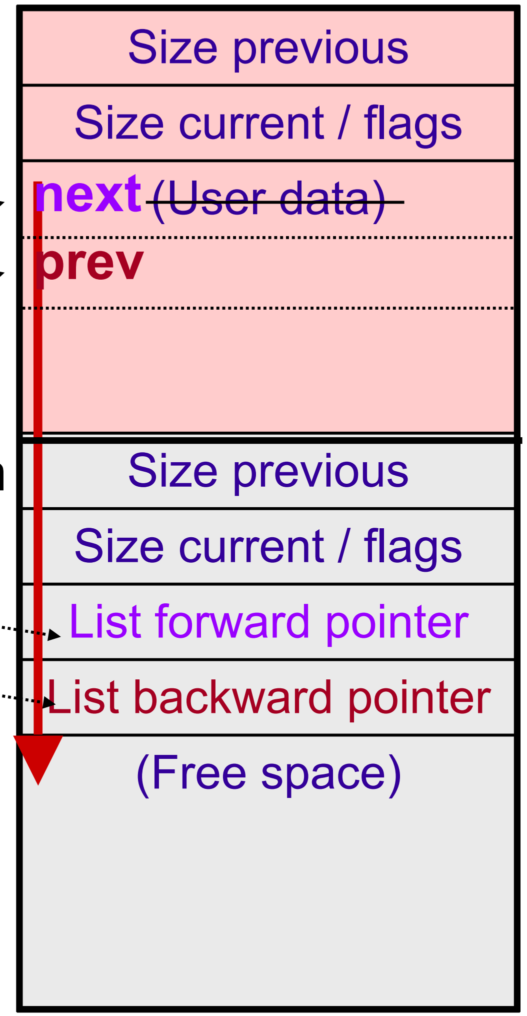
Heap overflow – advanced (II) …

Start with 2 blocks
(both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers
- 2nd step – program frees top block
  - The two blocks are merged running:

    next= bottom->next
    prev= bottom->prev

bottom



…

Heap overflow – advanced (II) …

Start with 2 blocks
(both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers
- 2nd step – program frees top block
  - The two blocks are merged running:

    next= **bottom->next**
    prev= **bottom->prev**
    next->prev = **prev**
    prev->next = **next**

bottom

…

Heap overflow – advanced (II) …

Start with 2 blocks
(both occupied)

- 1st step – overflow
  - Marks bottom block as free (changing flag)
  - Writes forward and backward pointers
- 2nd step – program frees top block
  - The two blocks are merged running:

    next= **bottom->next**
    prev= **bottom->prev**
    next->prev = **prev**
    prev->next = **next**
  - Attacker controls **prev** and **next** so can *write a value in any memory position*!
    - Last line: writes at backward pointer+few bytes the value in the forward pointer

bottom

…

Heap overflow – advanced (III)

- What can the attacker do by overwriting a 4-byte value in memory?
  - Modify security-wise relevant values in memory (e.g., flag indicating the user is authenticated)
  - Cause a jump to an arbitrary address in memory, by overwriting addresses of routines at:
    - Exit handlers
    - Exception handlers
    - Function pointers in the application
    - …

Taint Analysis

dangerous information flow

sensitive sink

```
3 void test(char *s) {
4  char buf[10];     // gcc stores extra space
5  strcpy(buf, s);   // does not check buffer's limit
6  printf("&s = %p\n&buf[0] = %p\n\n", s, buf);
7 }

1 main(int argc, char **argv) {
2   test(argv[1]);
8 }
```

tainted information

dangerous information flow:
- as it is, the program has a **vulnerability**, i.e., it is **exploitable**
- **not exploitable** if test of the length of *s is added between 3-4

Summary

- Buffer overflows
- Heap overflows
  - Basic, advanced
- Stack overflows
  - Stack smashing, code injection, arc injection, pointer subterfuge, off-by-one, read memory
- Integer overflows