Buffer Overflows

Ibéria Medeiros

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

- **Cisco Releases Fixes for Two Critical Flaws and Other Security Issues**
- **(March 8, 2018)**

- Cisco has released 22 security advisories to address issues in a variety of products. Two of the flaws are rated critical. The first is a hardcoded password in Cisco Prime Collaboration Provisioning (PCP) that a local attacker could use to attain root privileges. The issue affects only PCP 11.6, which was released in November 2016. The second is a Java deserialization issue in Cisco Secure Access Control System (ACS) that could be exploited remotely to execute arbitrary commands.

- **Google Patches Chrome Flaw**
- **(October 27, 2017)**

- Google has fixed a stack-based buffer overflow vulnerability in its Chrome browser that could be exploited to execute arbitrary code. The Chrome stable channel has been updated to 62.0.3203.75 for Windows, Mac, and Linux.

- **Linux Kernel Team Releases Patch for Flaw in ALSA**
- **(October 15 & 16, 2017)**

- A patch is available to fix a flaw in the Linux kernel. The use-after-free memory corruption vulnerability in ALSA (Advanced Linux Sound Architecture) could be exploited to execute code with elevated privileges.

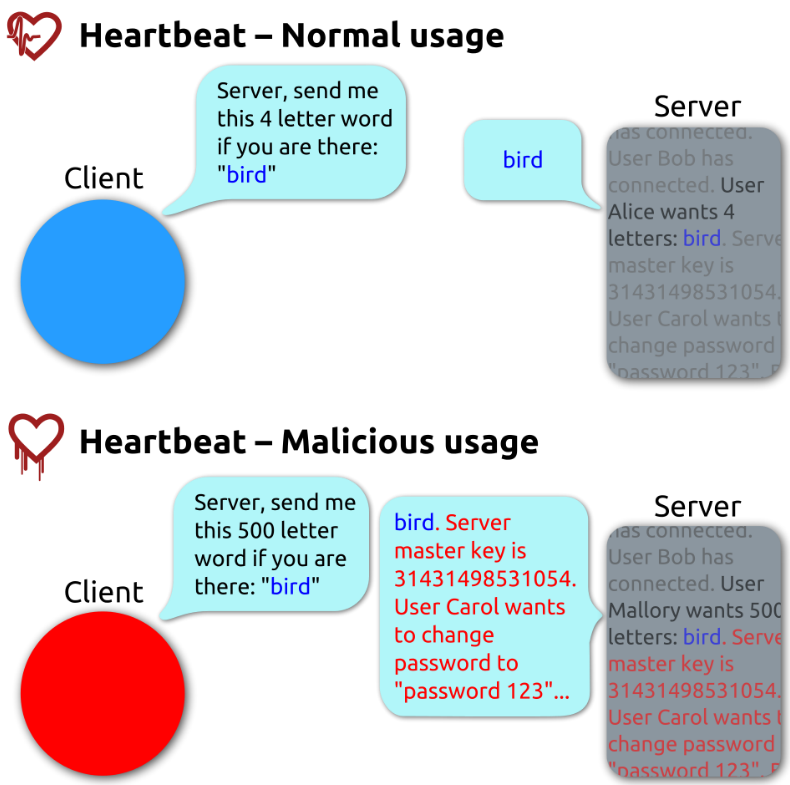
- **Read more in:**
- **- www.bleepingcomputer.com**: Patch Available for Linux Kernel Privilege Escalation

Patches for Linux Kernel Flaws (Dec 2016)

- Linux developers have released patches for a trio of vulnerabilities in the Linux kernel. The first and most serious is a **race condition** in the af_packet implementation function that local users could exploit to crash systems or run arbitrary code as root. The second is a **race condition** in the Adaptec AAC RAID controller driver that local users to crash a system. The third flaw is a **use after free vulnerability** that could be exploited to break the Linux kernel's TCP retransmit queue handling code and crash a server or execute arbitrary code. Patches available on all major Linux distributions.

Motivation

A vulnerability has been found in the heartbeat protocol implementation of TLS (Transport Layer Security) and DTLS (Datagram TLS) of OpenSSL. OpenSSL replies a requested amount up to **64kB of random memory content** as a reply to a heartbeat request. Sensitive data such as message contents, **user credentials, session keys and server private keys** have been observed within the reply contents. More memory contents can be acquired by sending more requests. The attacks have **not been observed to leave traces in application logs**.

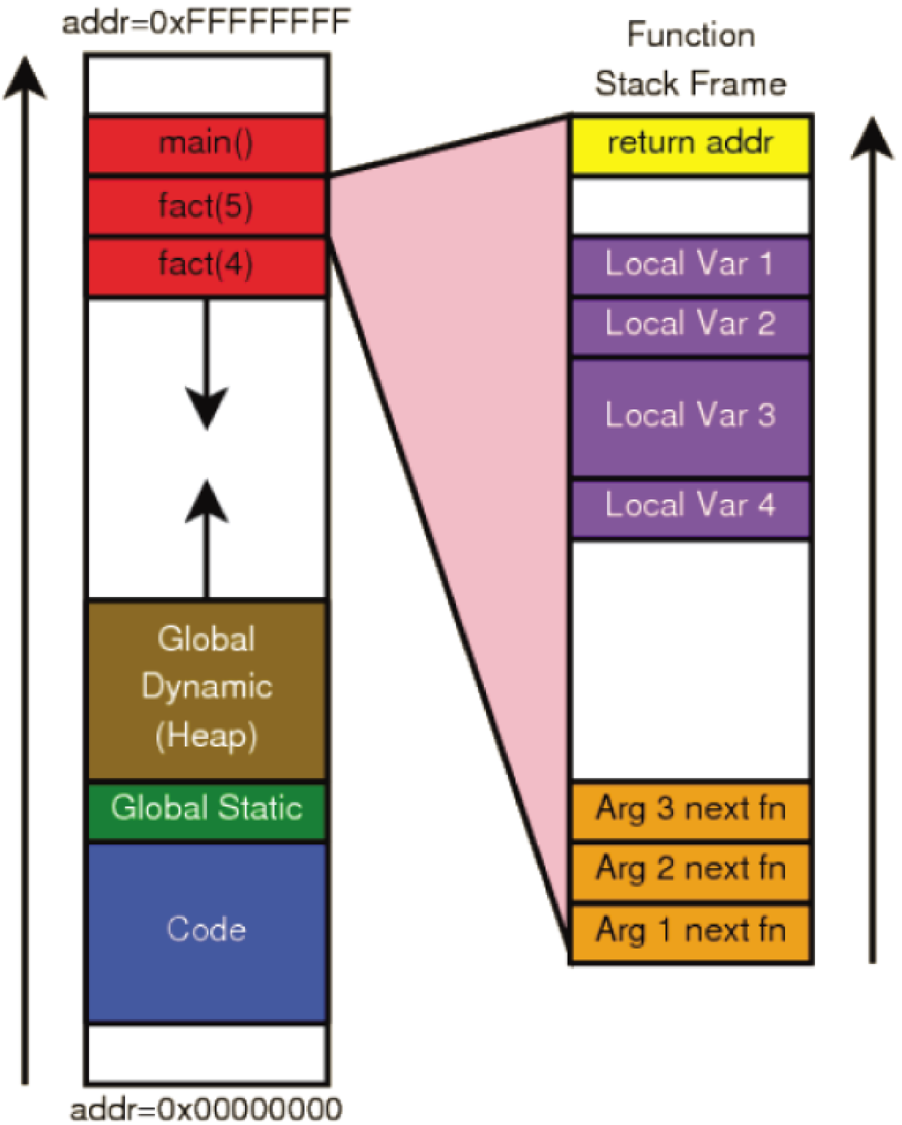


- Due to a buffer overflow in the implementation of Open SSL, 7 de April 2014

What is a BO?

- C programs store data in 4 places
  - ☞ stack – local variables
  - ☞ heap – dynamic memory (malloc, new)
  - ☞ data – initialized global variables
  - ☞ bss – uninitialized global variable

- Buffer
  - ☞ mem space with contiguous chunks of the same data type



What is a BO?

- C programs store data in 4 places
  - ☞ stack – local variables
  - ☞ heap – dynamic memory (malloc, new)
  - ☞ data – initialized global variables
  - ☞ bss – uninitialized global variable

- Buffer
  - ☞ mem space with contiguous chunks of the same data type

- **Buffer overflow** occurs when a program writes outside the allocated space for the buffer (or **buffer overrun** in Microsoft jargon), normally after the end

Cause

- Languages such as C and C++
  - ☞ the language does not verify if data overflows the limit of a buffer / array / vector
  - ☞ and, e.g., because programmers make 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 does a BO do?

- What happens when there is an accidental BO?
  - ☞ program becomes unstable
  - ☞ program crashes
  - ☞ 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 problem with such a bug is often hard
  - ☞ effects can appear several lines later

Why are BOs a security problem?

- Can be exploited intentionally and let the attacker execute its own code on the target machine
  - ☞ objective is usually to run code w/ superuser privileges

    … easy if server running with superuser privileges

    ... or afterwards use a **privilege escalation attack** to do the rest
  - ☞ important paper (mainstreamed these attacks): Aleph One, “Smashing the Stack for Fun and Profit”, Phrack 49-14.1996
- How do we prevent them?
  - ☞ Simple: always do bounds checking
  - ☞ Problems might arise only when you cannot control input

Wrong:

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

Right:

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

Note: fgets will add ‘\0’ at the end!

Functions to avoid in C

- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsprintf
- vscanf
- vsscanf
- streadd
- strecpy
- strtrns

This does not mean that they can not be used …
we simply need to be more careful! Example:

<u>Solution 1</u>

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

<u>Solution 2</u>

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

<u>Solution 3</u>

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

Internal BOs

- Occur in the buffers of a library function

- **char *realpath(const char *path, char *out_path)**
  - ☞ converts a relative path to the equivalent absolute path
  - ☞ problem: output string may be longer than the buffer provided
  - ☞ even if the size of the buffer is MAXPATHLEN, an internal buffer could be overrun!
- Other functions with similar problems
  - ☞ **syslog()**
  - ☞ **getopt()**
  - ☞ **getpass()**

NOTE: Current implementations of these functions probably no longer contain these problems!

Other risks

- Even “safe” versions of lib calls can be misused
  - ☞ for example, **strncpy**() has typically an undefined behavior if the two buffers overlap
  - ☞ for example, **strncpy**() does not add a ´\0´ if the original string is larger than the destination buffer
- **getenv()** – what is the size of the environment variable? One needs to be very careful when using the result from this function …

Lessons:

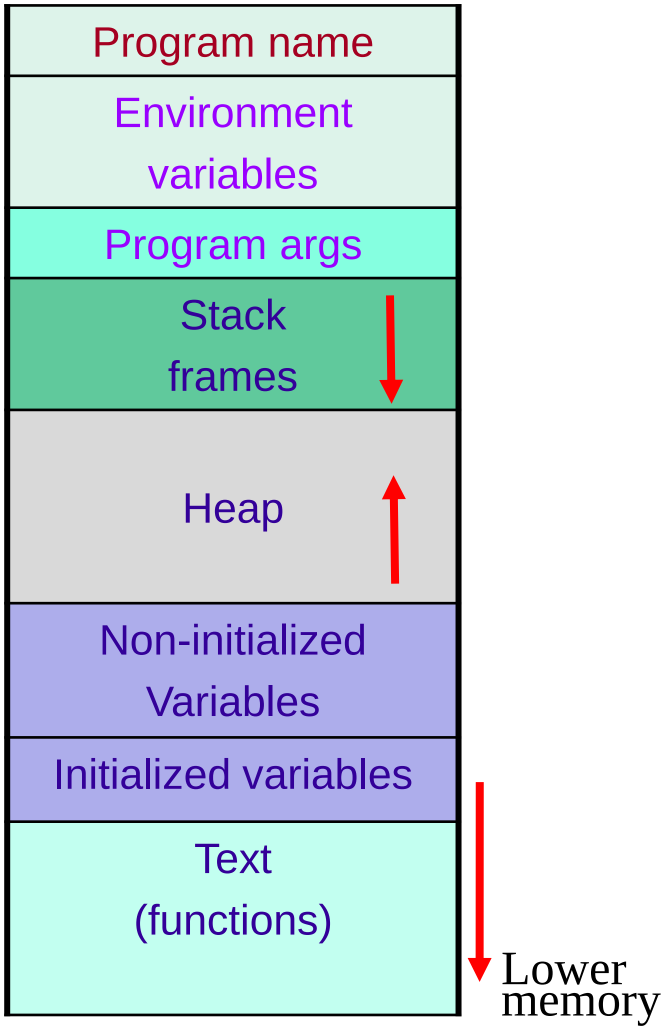
☞ Do not assume anything about someone else’s software

☞ NEVER TRUST INPUT !

Overflowing heap and stack

- Memory virtualization typically solved using one of two mechanisms
  - ☞ segmentation
  - ☞ pagination
- 80x86 processors support both

- 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 segment



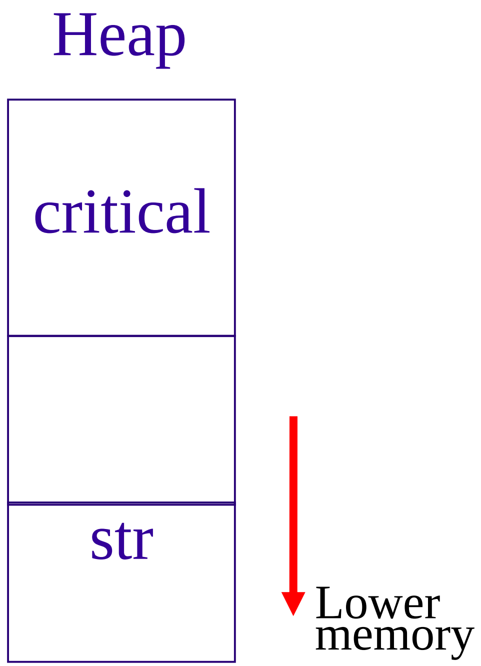
HEAP OVERFLOWS

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]); //vulnerab.
  printf("%s\n", critical);
}
```



Let us confirm that the heap is really organized as in the figure!

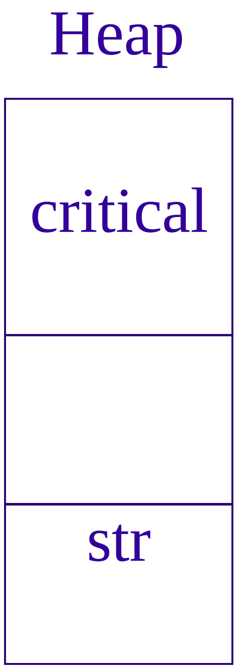
Heap overflow - basic (II)

```
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



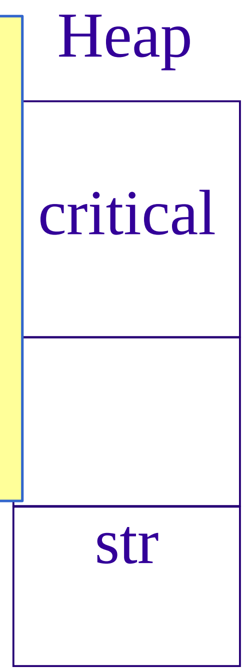
Heap overflow – basic (IV)

./a.out xyz1234567890123HEHEHE

Address of str is: 0x80497f0
Address of critical is: 0x8049800
0x804
0x804
0x804
0x804
0x804
0x804
0x804
0x804
0x80497f8: 6 (0x36)
0x80497f9: 7 (0x37)
0x80497fa: 8 (0x38)

0x80497fb: 9 (0x39)
0x80497fc: 0 (0x30)
0x8049805: E (0x45)
0x8049806: ? (0x0)
0x8049807: ? (0x0)
0x8049808: ? (0x0)
HEHEHE

NOTE: although this attack can be significant, we are limited to write to higher memory zones than the buffer, and probably not too far above the buffer (since we need to overwrite the whole memory in between the buffer with the overflow and the target and there might be unallocated memory pages!).



STACK OVERFLOWS

Stack overflow (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]);
}
```

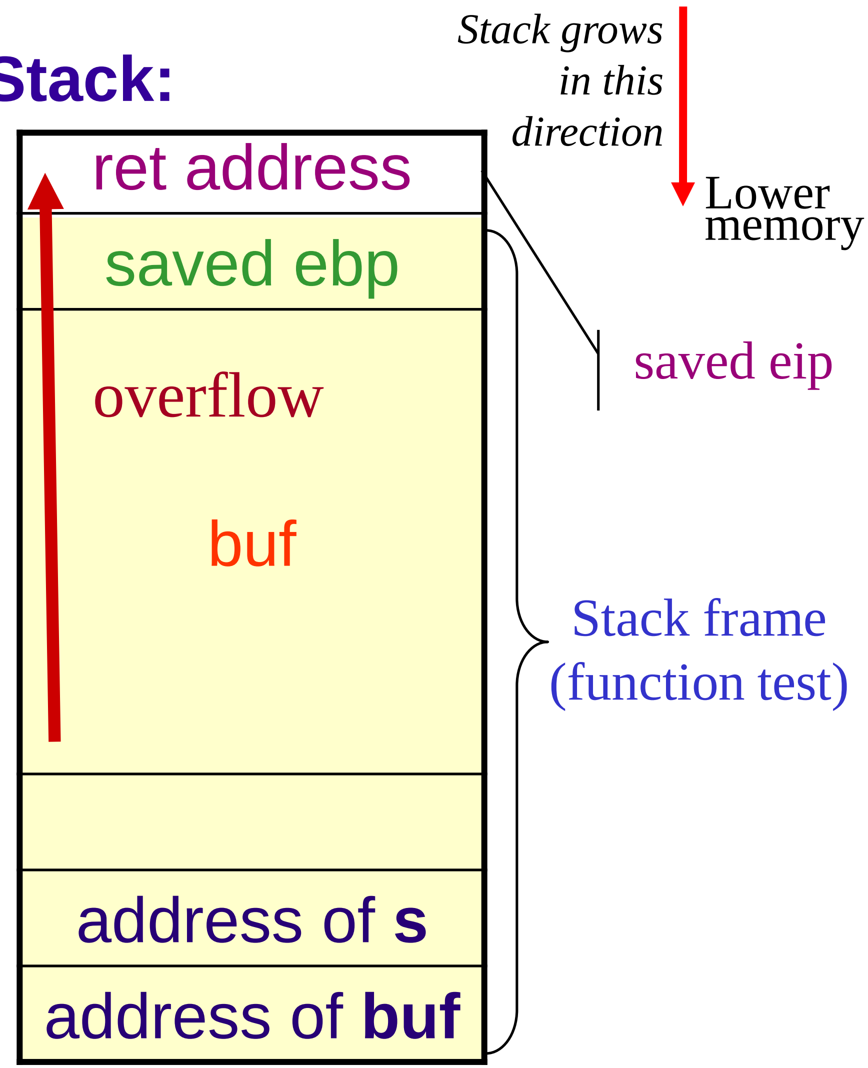
- The code is obviously vulnerable: inserts untrusted input in buffer without checking
- *gcc* compiles first to assembly...

Stack overflow (II)

get assembly: *gcc file.c -S*

```
test:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp          // buf
    subl $8, %esp
    pushl 8(%ebp)           // s
    leal -24(%ebp), %eax    // buf
    pushl %eax
    call strcpy
    ...
    ret  // jumps to ret address!!
main:
    …
    call test
```

Stack:



(Fast) Review of Assembly (32-bit x86; AT&T notation)

Register Category	Register Names	Purpose
General registers	EAX, EBX, ECX, EDX	Used to manipulate data
	AX, BX, CX, DX	16-bit versions of previous registers
	AH, BH, CH, DH, AL, BL, CL, DL	8-bit high- and low-order bytes of the previous registers
Segment registers	CS, SS, DS, ES, FS, GS	16-bit, holds the first part of a memory address; holds pointers to code, stack, and extra data segments
Offset registers		Offset related to segment registers
	EBP (extended base pointer)	Points to the beginning of the current stack frame
	ESP (extended stack pointer)	Points to the top of the stack
	ESI (extended source index)	Holds the data source offset in a operation using a memory block
	EDI (extended destination index)	Holds the destination data offset in a operation using a memory block
Special registers	EIP	Instruction pointer
	EFLAGS	Flags to track results of logic and state of CPU

(Fast) Review of Assembly (32-bit x86)

Instruction	Purpose
movl $51h, %eax	Copy data from the source to the destination; copy 51 (hex) to register EAX
add $51h, %eax	Add the source to the destination and store result in the destination; add 51 (hex) to register EAX
sub %ebx, %eax	Similar to addition … ; subtract EBX from EAX and place result in EAX
pushl %eax | popl	Push the argument to the top of the stack (pop does the opposite); Decrement the ESP by 4 bytes and store at (ESP) the value of EAX
xor %ebx, %eax	Bitwise exclusive or; XOR of EBX and EAX and place result in EAX
jne, je, jz, jnz, jmp	Jump the execution to another location depending on the eflag “zero flag”
call procedure	Calls the procedure; pushes EIP to the stack and jumps to procedure
ret	Return from a procedure; pops the return address to EIP and jumps to EIP
leave	Prepare stack before returning; equivalent to: movl %ebp, %esp; popl %ebp
inc %eax | dec %eax	Increment and decrement the value in a register
lea 4(%dsi), %eax	Load effective address into destination; load in EAX the address of [DSI+4]
int $0x80	Send a system interrupt signal to the processor

Stack overflow (III)

- Running the previous example:

```
$ ./a.out 12345
 &s = 0xbffffc11
 &buf[0] = 0xbffffa60

$ ./a.out 123456789012345678901234567890
Segmentation fault (core dumped)
```

Stack overflow (IV)

```
void cannot() {
  printf("Not executed!\n");
  exit(0);
}

void test(char *s) {
  char buf[10]; //gcc stores extra space
  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:
    pushl %ebp
    movl %esp, %ebp
    subl $24,%esp
    ...
    ret
main:
    ...
    call  test
    ...
```

Stack overflow (V)

- Can we write the address of “cannot” over the return address to main?

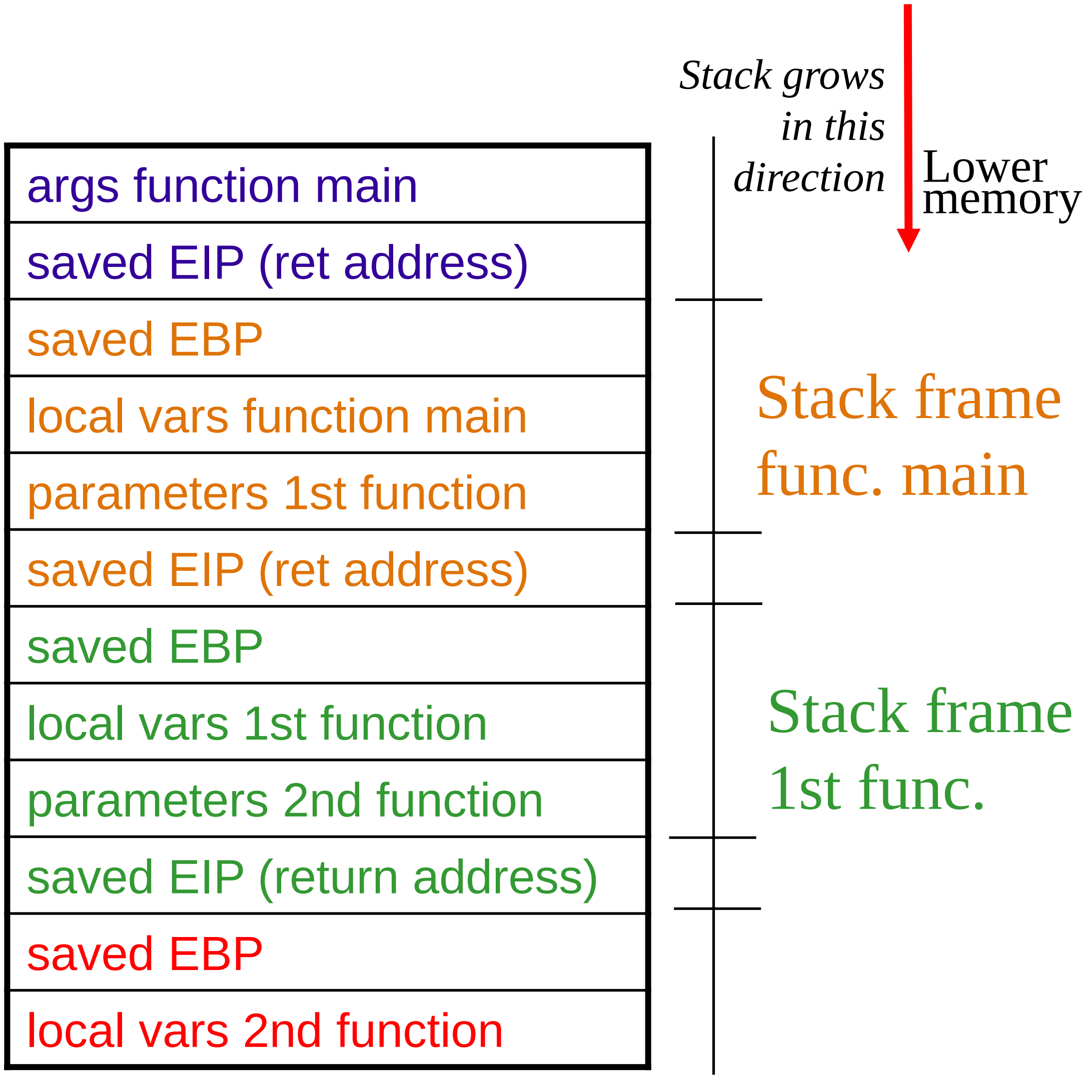
```
main() {
  int i;
  char *buf = malloc(1000);
  char **arr = (char **)malloc(10);
  for (i=0; i<28; i++) buf[i] = 'x';
  buf[28] = 0xd0;
  buf[29] = 0x84;
  buf[30] = 0x04;
  buf[31] = 0x08;
  arr[0] = "./stack_2";
  arr[1] = buf;
  arr[2] = 0x00;
  execv("./stack_2", arr);  // executes stack_2
                            // (previous slide)
}
```

```
$ ./call_stack_2
 &cannot = 0x80484d0
 &s = 0xbffffb00
 &buf[0] = 0xbffffa40

Not executed!
```

Stack overflow – stack layout

- SO attacks
- Means
  - ☞ Overflow local vars
  - ☞ Overflow saved EIP
- Effects
  - ☞ Modify state of progr.
  - ☞ Crash progr.
  - ☞ Execute code



Main Solutions for Protection

- **Address space layout randomization (ASLR)**
  - ☞ the starting address of the address space segments changes in each execution, preventing the pre-computation of
    - – particular addresses to be overwritten (e.g., a function pointer)
    - – location of a specific code
- **Data Execution Prevention (DEP)**
  - ☞ the stack pages cannot be executed, but only written/read
  - ☞ the code segment can be executed, but not written
- **Canaries**
  - ☞ put special (nondeterministic) values – **canaries** -- before (or after) the places we want to protect in memory
  - ☞ check that canaries have not been changed before accessing the protected memory

Example: Canaries
(same code as Stack overflow (I))

```
test:
      pushl   %ebp
      movl    %esp, %ebp
      subl    $40, %esp
      movl    8(%ebp), %eax
      movl    %eax, -28(%ebp)
      movl    %gs:20, %eax
      movl    %eax, -12(%ebp)
      xorl    %eax, %eax
      movl    -28(%ebp), %eax
      movl    %eax, 4(%esp)
      leal    -22(%ebp), %eax
      movl    %eax, (%esp)
      call    strcpy
      movl    -12(%ebp), %eax
      xorl    %gs:20, %eax
      je      .L2
      call    __stack_chk_fail
.L2:
      leave
      ret
```

gs is a segment register used for the thread local storage

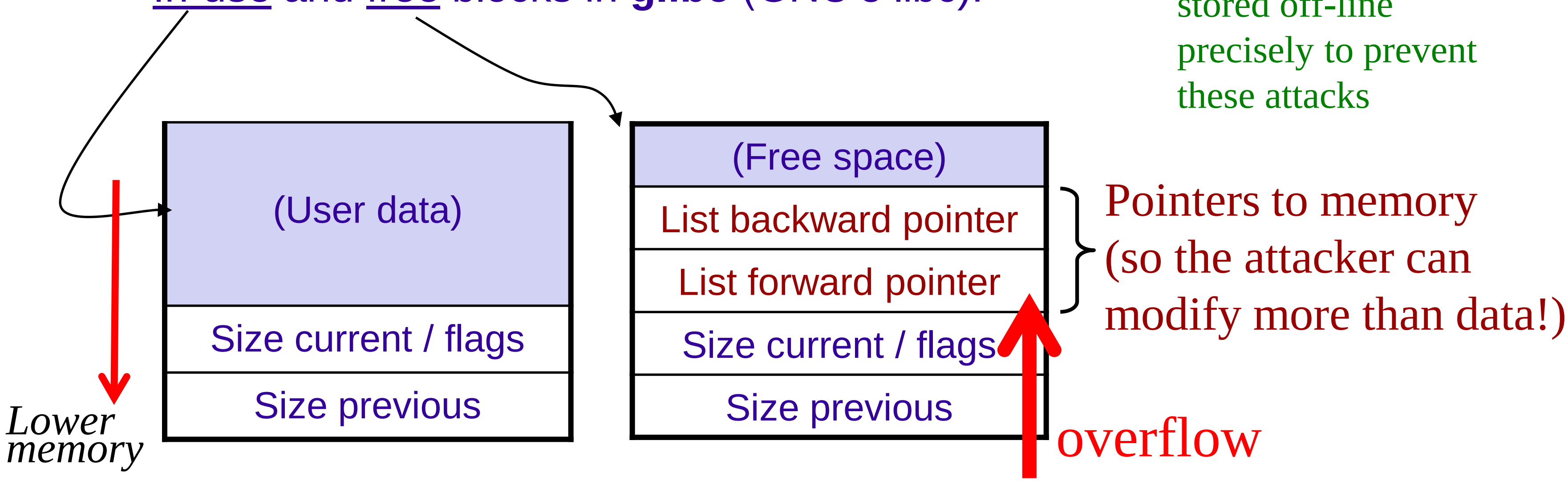
*In this case, %gs:20 stores a **canary** for this execution*

HEAP OVERFLOW ADVANCED

Heap overflow – advanced (I)

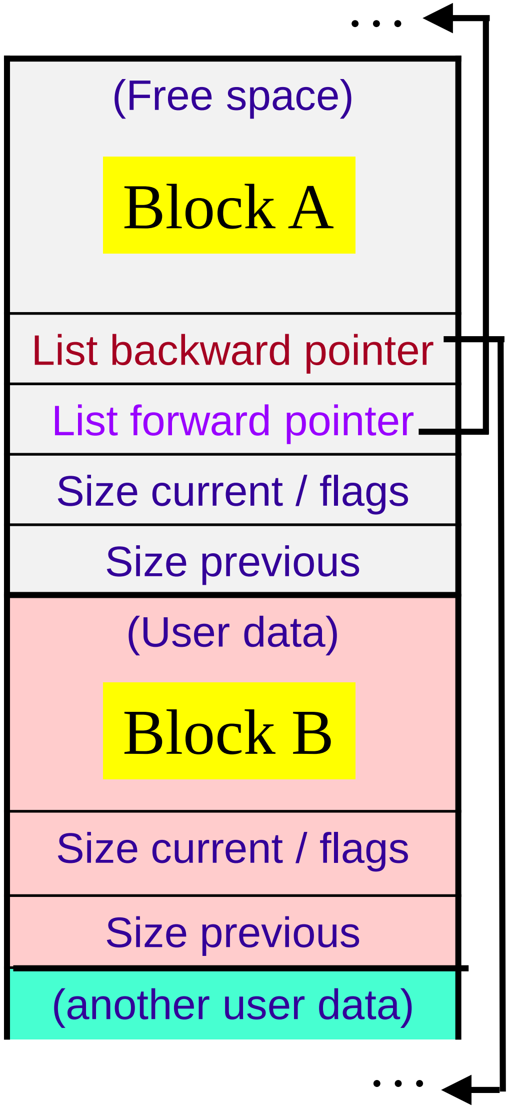
- Problem (for the hacker): heap implementations vary much
  - malloc: gets a block of data
  - free: frees a block (typically only marks it “free”)
- Free blocks usually chained using a doubly-linked list
- Usually blocks are stored with control data inline
  - size, link to next free, free/in-use,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

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



Heap overflow - advanced (II)

- Assume that the block above (block A) is free, and we want to overflow the block below (block B)
- When the program frees the bottom block (B), it is typically merged with the contiguous free blocks to create a bigger free block by:
  1. the already free buffer (A) is removed from the free list
  2. the control information in the new free block (e.g., size) (B) is updated to represent the merge of two/three free blocks
  3. the new free block is inserted in the free list

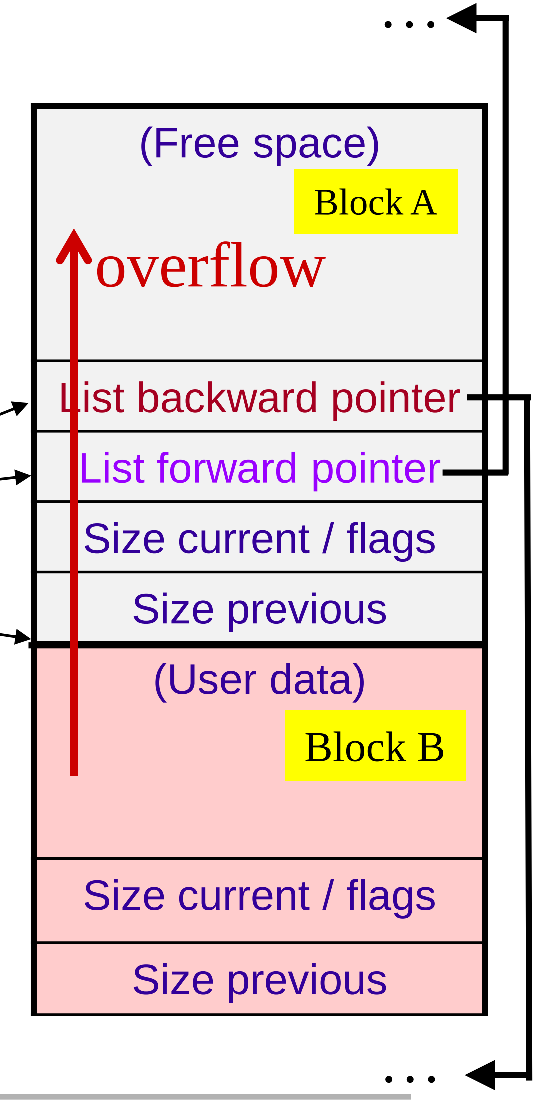


Notice that in typical implementations of malloc, the top block does not even have to be free since we can make it look free with the overflow!

Let us look in more detail to the **1st step!**

Heap overflow – advanced (III)

- First – overflow (on B)
  - ☞ marks top block as free (changing flag)
  - ☞ writes over forward and backward pointers
- Second – program frees bottom block (B)
  - ☞ The top block (A) is removed from the list by
    1 ListElement ***next**= **top**->**next**
    2 ListElement ***prev**= **top**->**prev**
    3 **next**->prev = **prev**
    4 **prev**->next = **next**
  - ☞ Attacker controls **prev** and **next** so can *write a 4 byte value in any memory position*!
    - Line 3: writes at (forward pointer + few bytes) the value in the backward pointer



Heap overflow – advanced (IV)

- 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
    - The Procedure Linkage Table (PLT)
    - …

Use-After-Free Vulnerability

```
#define BUFSIZE   8
int main(int argc, char **argv) {
    char *b1, *b2, *b3, *b4;

    b1 = (char *) malloc(BUFSIZE);
    b2 = (char *) malloc(BUFSIZE);
    free(b2);

    b3 = (char *) malloc(BUFSIZE);
    b4 = (char *) malloc(BUFSIZE);
    sprintf(b3, "ola!");
    strcpy(b2, argv[1]);  // b2 used after free

    printf("b1 = %p; b2 = %p; b3 = %p; b4 = %p\n",
        b1, b2, b3, b4);
    printf("b2 = %s; b3 = %s; b4 = %s\n", b2, b3, b4);

    free(b1); free(b3); free(b4);
}
```

b3 gets the same memory region as b2

Use-after-free
b2 possibly ends up changing b3, corrupting its content!

Can also cause problems with free(), depending if argv[1] was large

Use-after-free Vulnerability

- Occurs when a program continues to use a pointer after it has been freed, with causes related to
  - ☞ error conditions and other exceptional circumstances
  - ☞ confusion over which part of the program is responsible for freeing the memory
- Impact
  - ☞ integrity: use of previously freed memory may **corrupt valid data**, if the memory area was allocated and used properly elsewhere
  - ☞ availability: if chunk consolidation occurs after the use of previously freed data, the process **may crash** when invalid data is used as chunk information
  - ☞ arbitrary execution: if malicious data is entered before chunk consolidation can take place, it may be possible to take advantage of the malloc ***write-what-where primitive*** to execute **arbitrary code**

STACK OVERFLOW ADVANCED

Stack overflow – practical aspects

- How do we find out the place of the return address that has to be overwritten by the BO?
  - ☞ without source code
    - – trial & error (see for example “Smashing the Stack for Fun and Profit”) **or**
    - – reverse engineer the code

- How do we do something useful?
  - ☞ inject attack code – **shell code**
  - ☞ in Unix, e.g., make program give a shell: /bin/sh
  - ☞ in Windows, e.g., download a remote admin tool like BackOrifice or Sub7, or bot code

Code injection

- In Unix, the code to span a shell can be only:

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

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

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

Difficulties with code injection

- Lack of space: reduce code <u>or</u> provide at an earlier time the code so that it is available when needed (e.g., environment var)
- Shell code includes zeros
  - ☞ (functions like strcpy() stop in the first zero)
  - ☞ Substitute places where zeros appear with equivalent code
  - ☞ Example: <u>movl $0, %eax</u> equivalent to <u>xorl %eax, %eax</u>
- Discover address where code is injected
  - ☞ The return address has to be superseded with this address
- Escape several forms of protection (e.g., non executable stack; stack canaries)

… next →

Arc injection *or* return-to-libc

- **Difficulty**: the stack **cannot** be executed
- Insert a new ***arc*** in the program control-flow *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

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

  - ☞ Register *R* has a pointer to an attacker supplied string (the *progname*)
    - – registers are reused and can point to a buffer in the stack
  - ☞ *Return address* in the stack (saved EIP) is set to *target,* causing the processor to jump there; this address is known if *libc* is loaded in the same place

Pointer subterfuge

- **Difficulty:** circumvent protections against BOs, where the return address in the stack is protected with a canary
- In general the exploit involves ***modifying a pointer***
- To some extent, the actual implementation depends on how the compiler lays out local variables and parameters

- Four example techniques
  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 overflow
     - Modify the C++ *virtual function table* associated with a class

1. Function-pointer clobbering

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

The return address from func does not have to be changed!

```
void func(void * arg, size_t len) {
   char buf[100];
   void (*f)() = …;      /* function pointer */
   memcpy(buf, arg, len); /* buffer overrun! */
   f();
   /* ... */
   return;
}
```

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

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 an *arbitrary memory write*

The return address from func does not have to be changed!

```
void(*f)() = … ;

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

Notice that the variable with function pointer **f** is not local, thus not prone to function pointer clobbering. But with the 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 (e.g., access violation), 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)
  - ☞ An exception is caused in some way (e.g., writing over all the stack causes an exception when its base is overwritten)

<u>NOTE</u>: Some validity checking of the SEH is done since Windows Server 2003, making this attack more difficult
<u>NOTE1</u>: In Linux there are also used lists of pointers, either in the heap or stack, that could also be exploited in a similar way

4. Virtual pointer overflow

- Virtual functions are used in C++ to allow a child class to redefine a function inherited from the mother class
- Most C++ compilers use a *virtual method table (VTBL)* associated with each class
  - ☞ VTLB is an array of pointers to methods
  - ☞ An object has in its header a *virtual pointer (VPTR)* to its class VTBL
  - ☞ An attacker can overrun the VPTR of an object with a pointer to a mock VTBL (with pointers to attacker supplied code, libc,…)

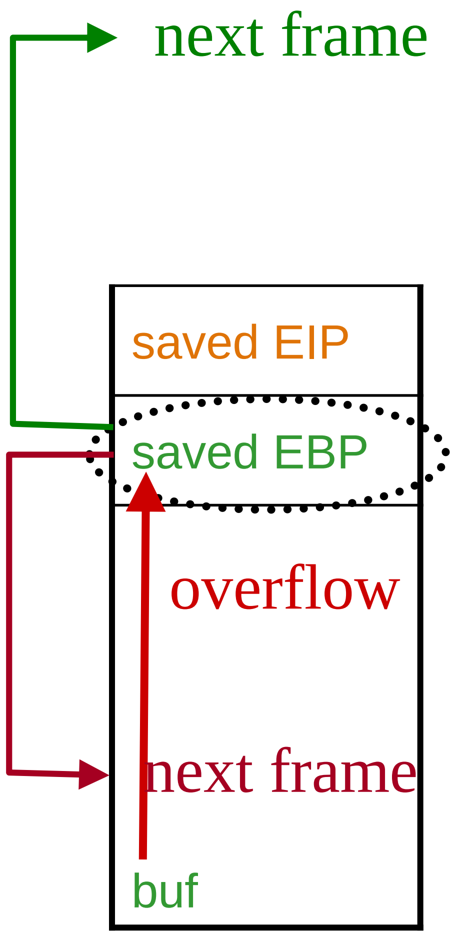
```
void method(void * arg, size_t len)
{
  char *buff = new char[100];
  C *ptr = new C;
  memcpy(buff, arg, len); // buffer overrun!
  ptr->vf();              // call to a virtual function
  return;
}
```

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);
}
```

- BO of (‘\0’) if *user* has (1024 chars + ‘\0’)
  - ☞ 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 arguments can be modified…
    - ➔ when the function returns, ESP becomes equal to EBP, and then the return address is poped to EIP …



Return-Oriented Programming (ROP)

- Think about the various forms of code in a process address space
  - program
  - libraries
- Think about the many places where there are returns in that code
- Think about the code immediately **before** the return

- NOTE: assume that there is **no** stack canary protection or ASLR

```
int global;
...
int funct1(int a, int b) {
    ...
    a = 1;
    return 0;            (1)
    ...
    c = a + b;
    return c;            (2)
    ...
}
void funct2(float aux){
    ...
    d = 2;
    global = d * d;      (3)
    return;
}
```

ROP (2)

- Now, select and reorganize those pieces of code in order to get a relevant program (forgetting for now the return statement)
- Recall that registers will be used across functions, for instance, in math operations

```
a = 1;
return 0;
```
(1)

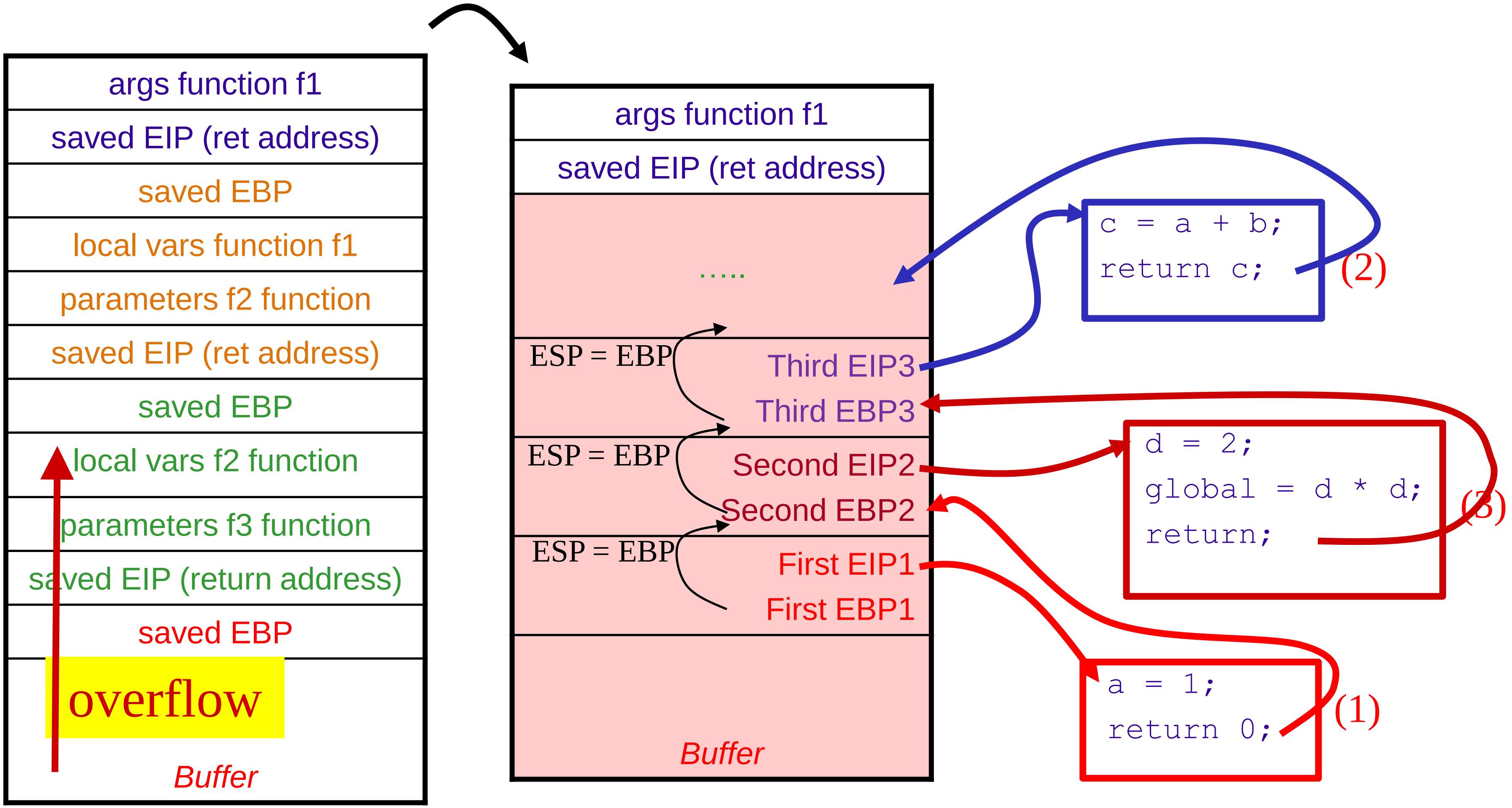
```
d = 2;
global = d * d;
return;
```
(3)

```
c = a + b;
return c;
```
(2)

< = >

```
x = 1;
y = 2;
z = x + y;
```

ROP (3)

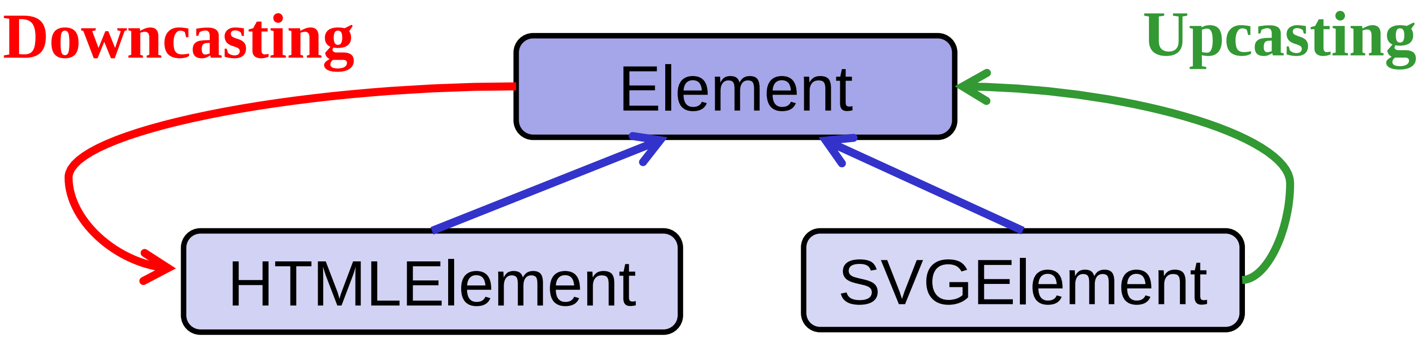


leave	# equivalent	movl %ebp, %esp;	popl %ebp
ret	# equivalent	popl %eip,	jmp %eip

DOWNCASTING OVERFLOWS

C++: Upcasting and downcasting

- Upcasting: from a derived class to its parent class
- Downcasting: from a parent class to one of its derived classes



Upcasting is always safe, but downcasting is not!

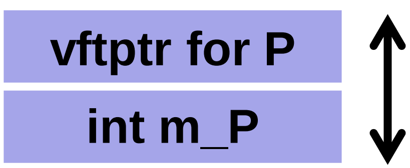
Why is downcasting unsafe? (1)

A class P with a destructor and a integer local variable

```
class P {
  virtual ~P()
     { /* do nothing */ }
  int m_P;
};
```

A class D that inherits from P, with a destructor and a integer local variable

```
class D : public P {
  virtual ~D()
       {/* do nothing */ }
  int m_D;
};
```



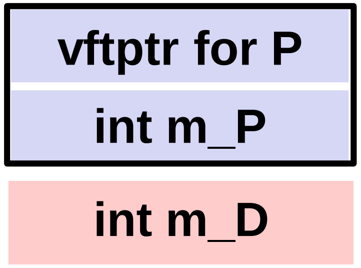
Access scope of a pointer to P

Access scope of a pointer to D

vftptr: virtual function table pointer

Why is downcasting unsafe? (2)

```
P *pS = new P();
D *pD = static_cast<D*>(pS);
pD->m_D = 1;
```



Out of bound memory access

Bad-casting occurs: P is a parent class of D

Attack example: imagine that

(i) m_D is actually a large amount of memory (e.g., a buffer);

(ii) m_D is on top of a vftptr of another object V, which you can overwrite with an arbitrary value;

(iii) the corresponding method of that object V is accessed

INTEGER OVERFLOWS

Integer overflows – basics

- The semantics of integer-handling is complex and programmers often don’t know the details
  - ☞ Can appear 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#)
  - ☞ First two also in Blockchain smart contract (e.g., Solidity)
- Five example 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)
- Case we consider: C99 ILP32 (int=long=pointer=32bits)

1. Overflow

- Result of expression exceeds maximum value of the type

- Probably the most common integer overflow form

- Typically, a overflow is handled by the system as

  if (x != overflow)   x = x

  else   x = (x mod MAX_SIZE_TYPE_x)

```
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;
   }
  }
}
```

If overflow of x*y, then not enough memory is allocated!

2. Underflow

- 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 since only with subtraction, never with other operations

Netscape JPEG comment length vulnerability:

```
void vulnerable(char *src, size_t len){
  size_t len_real;       // unsigned data type
  char *dst;
  if (len < MAX_SIZE) {
      len_real = len - 1; // no need to save '\0'
      dst = (char *) malloc(len_real);
      memcpy(dst, src, len_real);
  }
}
```

If len=0, then len_real = FFFFFFFF, and malloc may return NULL

3. Signedness error

- A signed integer is interpreted as unsigned or vice-versa
  - ☞ Negative number interpreted as positive → the sign bit (1) is interpreted as 2^{31}

Signed integers are typically represented in two's complement

MSB = 1 means negative number

MSB = 0 means positive number

Linux kernel XDR vulnerability:

```
1 void vulnerable(char *src,
                           size_t len){
2  int lReal;
3   char *dst;

4    if (len > 1) {
5     lReal = len - 1;
6     if (lReal < MAX_SIZE) {
7      dst= (char*)malloc(lReal);
8      memcpy(dst, src, lReal);
9     }
10   }
11 }
```

Line 5: lReal is 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)

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

SSH CRC-32 compensation attack detector vulnerability:

```
void vulnerable(char *src,
                   unsigned int len) {
  unsigned short lReal;
  char *dst;

  lReal = len;
  if (lReal < MAX_SIZE) {
   dst= (char *) malloc(lReal);
   strcpy(dst, src);
  }
}
```

Portability ILP32 -> LP64

Bibliography

- M. Correia, P. Sousa, Segurança no Software, FCA Editora, 2017 (see chapter 5)

Other references:

- J. Foster, V. Osipov, N. Bhalla, N. Heinen, Buffer Overflow Attacks: Detect, Exploit, Prevent, Syngress, 2005
- B.Lee, C.Song, T.Kim, W.Lee, Type Casting Verification: Stopping an Emerging Attack Vector, Usenix Security Symposium, 2015