Buffer Overflows

Ibéria Medeiros

Departamento de Informática

Faculdade de Ciências da Universidade de Lisboa

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

- **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.

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

- **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.

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

- **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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

Patches for Linux Kernel Flaws (Dec 2016)

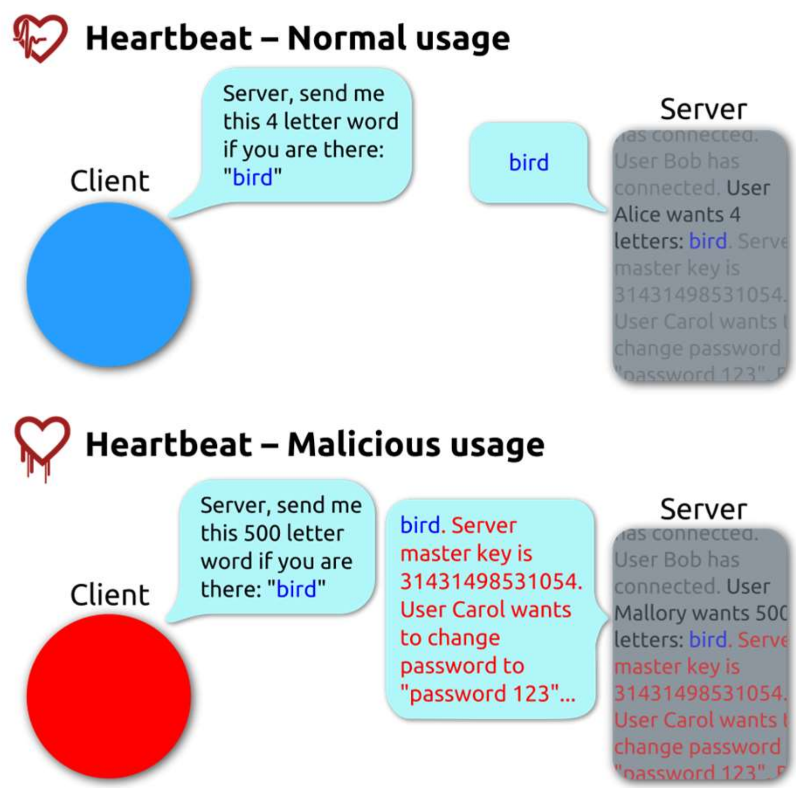
- Linux developers have released patches for a trio of vulnerabilities in the Linux kernel. The first and most serious is a **race condition** in the af_packet implementation function that local users could exploit to crash systems or run arbitrary code as root. The second is a **race condition** in 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.

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

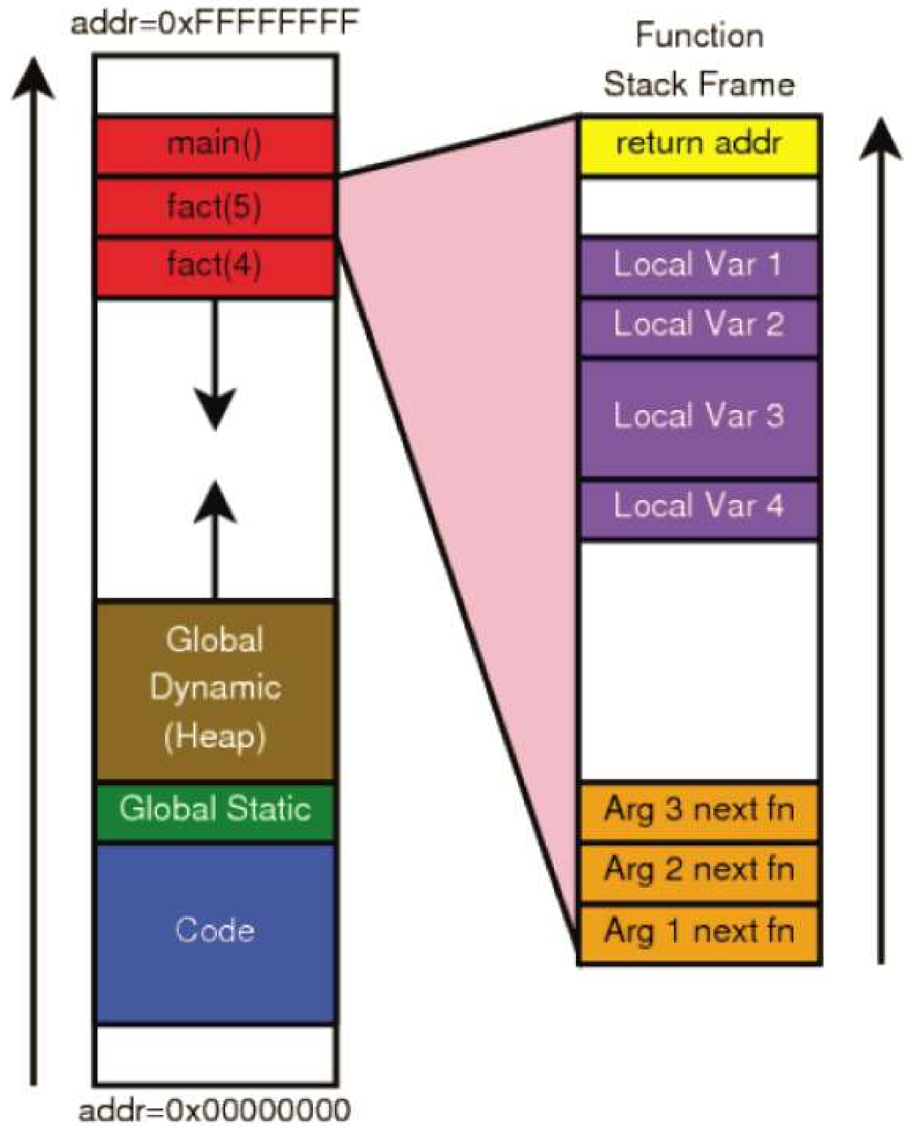


© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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



© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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!

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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)
```

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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!

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

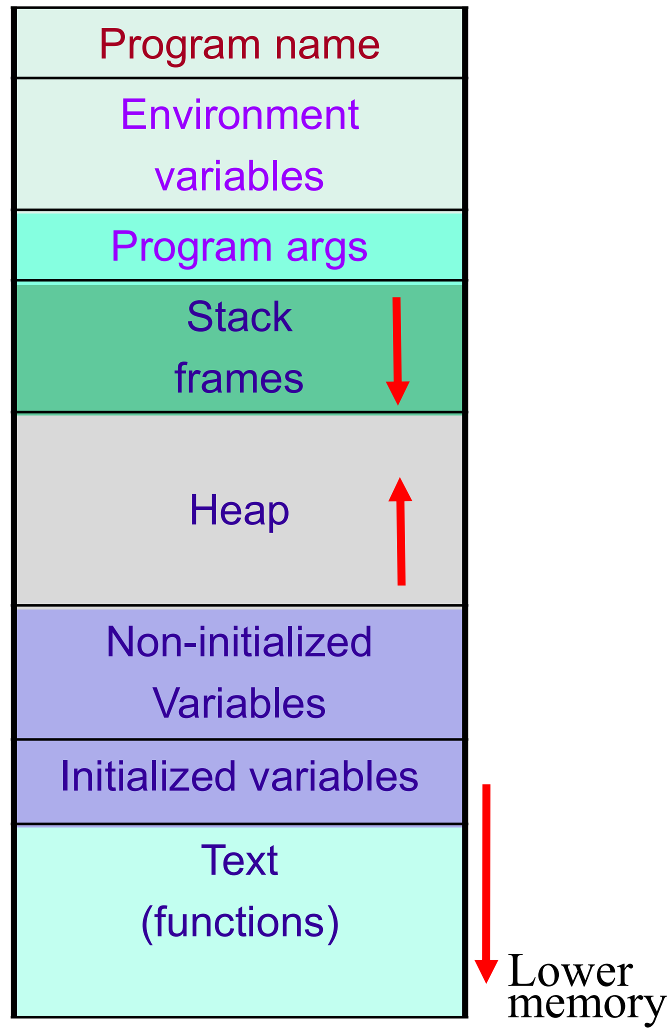
☞ NEVER TRUST INPUT !

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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



© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

HEAP OVERFLOWS

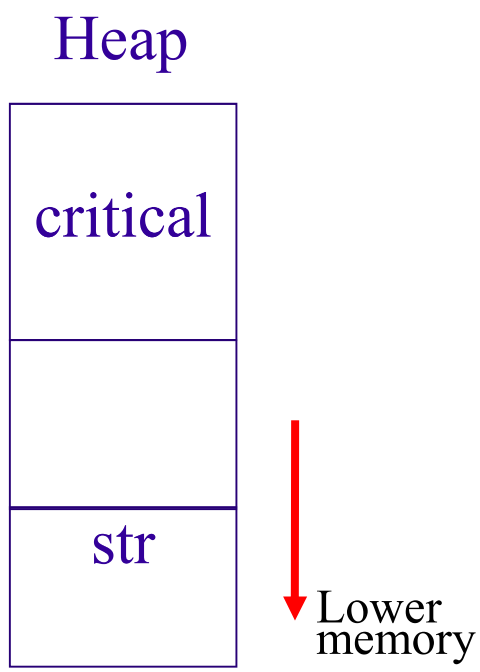
© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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!

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

Heap overflow – basic (II)

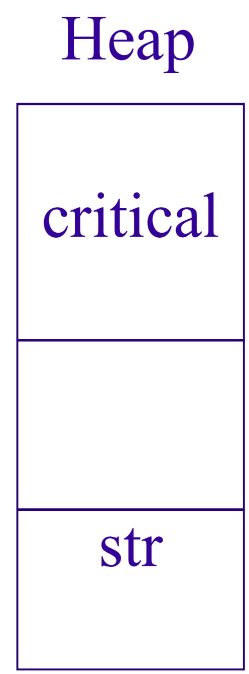
```
main(int argc, char **argv) {
  int i;
  char *str = (char *)malloc(4);
  char *critical = (char *)malloc(9);
  char *tmp;
  printf("Address of str is: %p\n", str);
  printf("Address of critical is: %p\n", critical);
  strcpy(critical, "secret");
  strcpy(str, argv[1]);
  tmp = str;
  while(tmp 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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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



© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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!).

Heap

critical

str

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

STACK OVERFLOWS

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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...

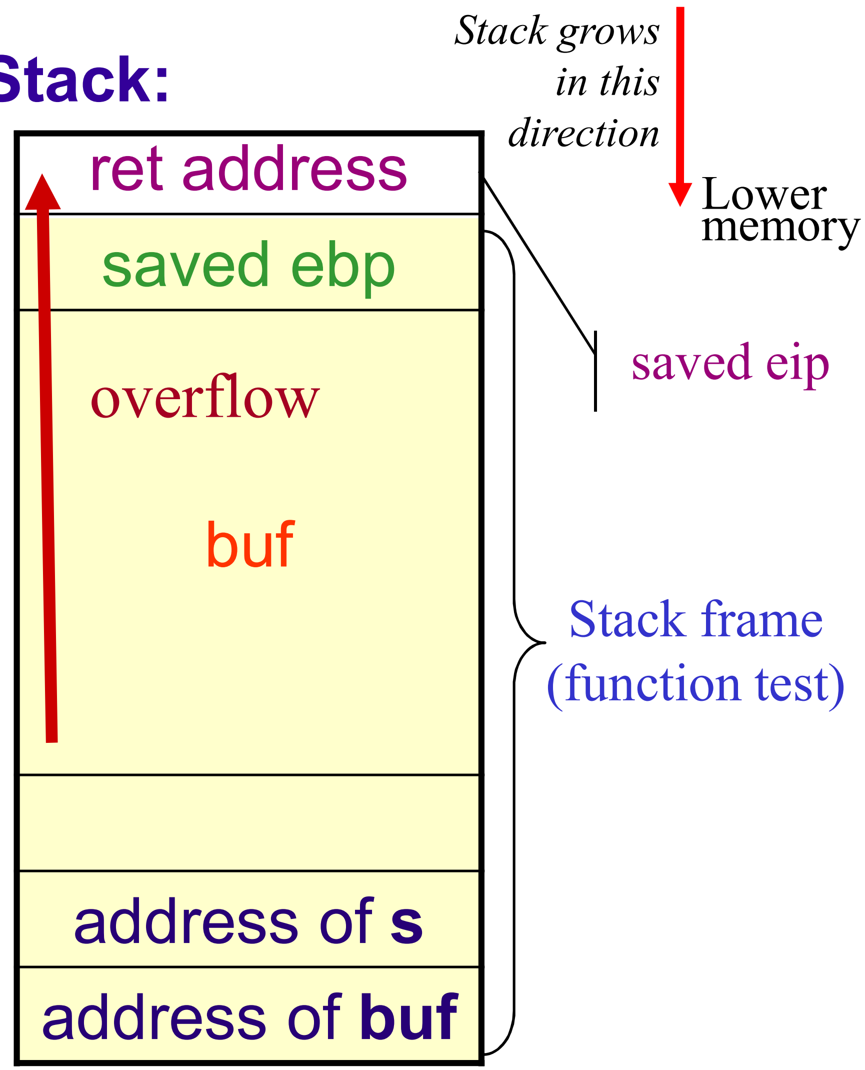
© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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:



© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

(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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

(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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

Stack overflow (III)

- Running the previous example:

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

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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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
    ...
```

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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)
}
```

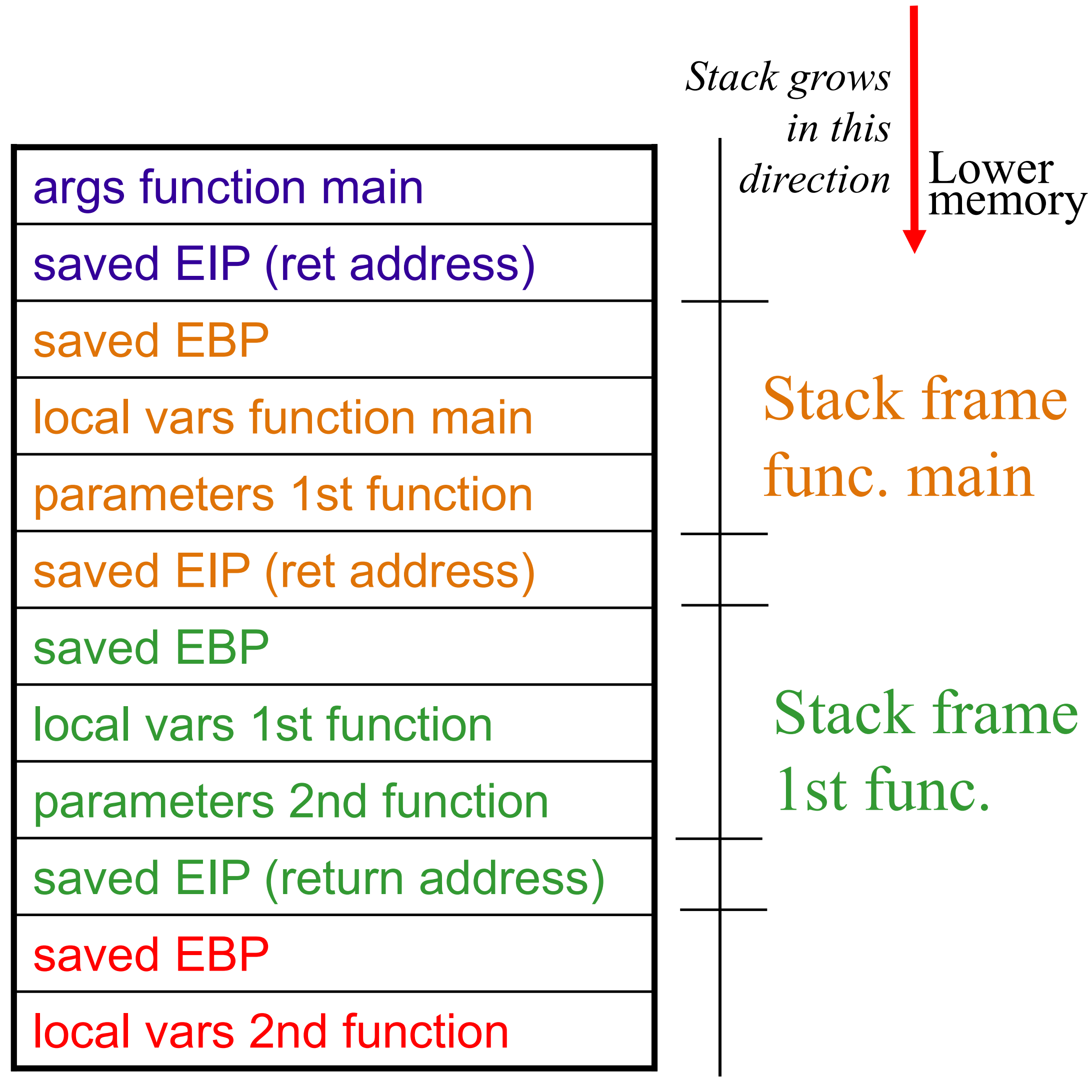
```
$ ./call_stack_2
 &cannot = 0x80484d0
 &s = 0xbffffb00
 &buf[0] = 0xbffffa40

Not executed!
```

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

Stack overflow – stack layout

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



© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.

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*

© 2018 Nuno Ferreira Neves, Ibéria Medeiros. All rights reserved. Reproduction only by permission.