



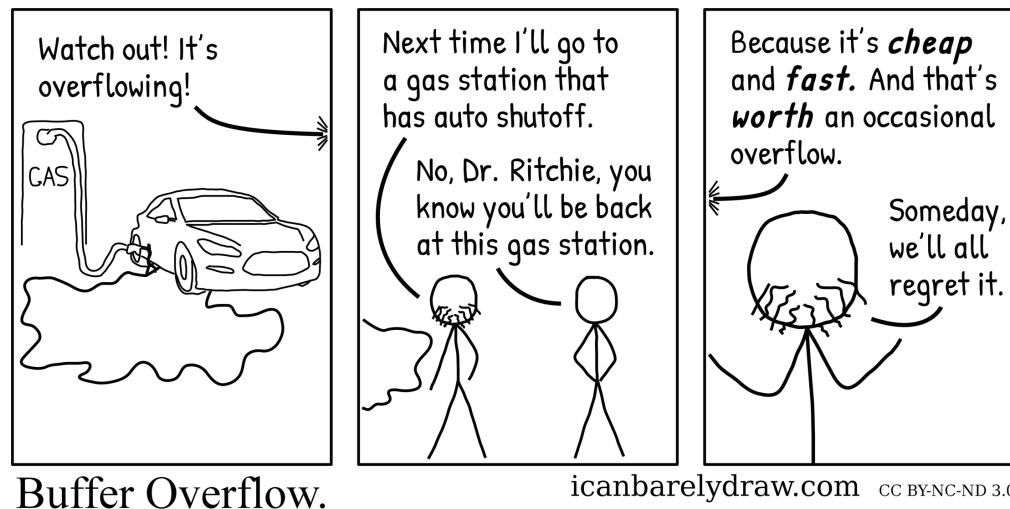
Mestrado em Engenharia Informática (MEI) Mestrado Integrado em Engenharia Informática (MiEI)

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

Engenharia de Segurança

Tópicos de Segurança de Software

- Vulnerabilidade de *Buffer overflow*



Buffer Overflow

The CWE Top 25

Below is a brief listing of the weaknesses in the 2021 CWE Top 25, including the overall score of each.

Rank	ID	Name	Score	2020 Rank Change
[1]	CWE-787	Out-of-bounds Write	65.93	+1

https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	CWE-787	Out-of-bounds Write	64.20	62	0

Buffer Overflow

CWE-787: Out-of-bounds Write

Weakness ID: 787

Abstraction: Base
Structure: Simple

View customized information:

Conceptual

Operational

Mapping
Friendly

Complete

Custom

▼ Description

The product writes data past the end, or before the beginning, of the intended buffer.

▼ Extended Description




Typically, this can result in corruption of data, a crash, or code execution. The product may modify an index or perform pointer arithmetic that references a memory location that is outside of the boundaries of the buffer. A subsequent write operation then produces undefined or unexpected results.

▼ Alternate Terms

Memory Corruption: Often used to describe the consequences of writing to memory outside the bounds of a buffer, or to memory that is invalid, when the root cause is something other than a sequential copy of excessive data from a fixed starting location. This may include issues such as incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release, etc.

▼ Relationships

▼ Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name
ChildOf		119	Improper Restriction of Operations within the Bounds of a Memory Buffer
ParentOf		121	Stack-based Buffer Overflow
ParentOf		122	Heap-based Buffer Overflow
ParentOf		123	Write-what-where Condition
ParentOf		124	Buffer Underwrite ('Buffer Underflow')
CanFollow		822	Untrusted Pointer Dereference
CanFollow		823	Use of Out-of-range Pointer Offset
CanFollow		824	Access of Uninitialized Pointer
CanFollow		825	Expired Pointer Dereference

<https://cwe.mitre.org/data/definitions/787.html>

Buffer Overflow

Variable 1							Var 2	
0	0	0	0	0	0	0	3	9
Variable 1							Var 2	
O	V	E	R	F	L	O	W	9

- *Buffer* – espaço temporário em memória destinado a guardar dados durante a execução do programa;
- *Buffer overflow*
 - Ocorre quando os dados escritos para um buffer têm um tamanho maior que o tamanho do buffer, e devido a insuficientes verificações, são escritos para localizações de memória adjacentes, podendo causar uma falha de programa ou a criação de uma vulnerabilidade que os atacantes possam explorar.
 - Reescreve/sobre-escreve posições de memória adjacentes, incluindo outros buffers, variáveis e código do programa (podendo mudar o fluxo do programa).
- Considerada a “bomba atômica” da indústria de software, o *buffer overflow* é uma das mais persistentes vulnerabilidades de software e onde as tentativas de ataque são mais frequentes.

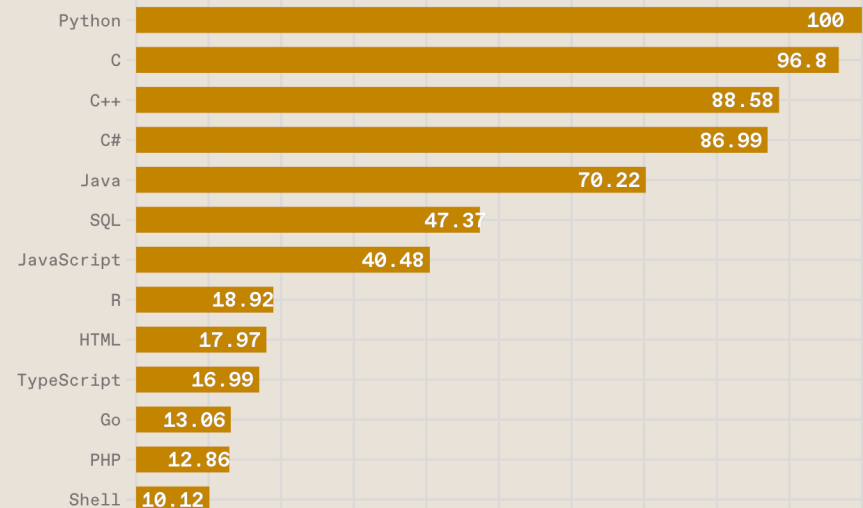
Buffer Overflow

- Risco
 - Escrever fora dos limites da memória alocada pode corromper os dados, parar o programa ou, causar a execução de código malicioso.
 - O **C** e o **C++** são particularmente vulneráveis ao *buffer overflow* e ainda são duas das quatro linguagens de programação mais utilizadas e mais pedidas pelos empregadores (por uma questão de performance) e nas quais são escritas programas críticos (S.O., X, IIS, shell, Apache httpd, MySQL, MS SQL server, *Mars rover*, sistemas controlo industrial, sw automóveis, IoT, ...);
 - O Java, por desenho, evita o *buffer overflow*, verificando os limites do buffer e prevenindo o acesso para além desse limite.

Top Programming Languages 2022

Click a button to see a differently weighted ranking

Spectrum Jobs Trending



Buffer Overflow

CVE ID	Vulnerability type	Publish Date	CVSS Score	Description
CVE-2021-1770	Improper Restriction of Operations within the Bounds of a Memory Buffer	09/08/2021	9.8	A buffer overflow may result in arbitrary code execution. This issue is fixed in macOS Big Sur 11.3, iOS 14.5 and iPadOS 14.5, watchOS 7.4, tvOS 14.5. A logic issue was addressed with improved state management.
CVE-2020-3807	Buffer Copy without Checking Size of Input	03/25/2020	9.8	Adobe Acrobat and Reader versions 2020.006.20034 and earlier, 2017.011.30158 and earlier, 2017.011.30158 and earlier, 2015.006.30510 and earlier, and 2015.006.30510 and earlier have a buffer overflow vulnerability. Successful exploitation could lead to arbitrary code execution
CVE-2020-10837	Improper Restriction of Operations within the Bounds of a Memory Buffer	03/24/2020	9.8	An issue was discovered on Samsung mobile devices with P(9.0) and Q(10.0) (with TEEGRIS) software. The Esecomm Trustlet allows a stack overflow and arbitrary code execution. The Samsung ID is SVE-2019-15984 (February 2020).
CVE-2020-4207	Buffer Copy without Checking Size of Input	01/28/2020	9.8	IBM Watson IoT Message Gateway 2.0.0.x, 5.0.0.0, 5.0.0.1, and 5.0.0.2 is vulnerable to a buffer overflow, caused by improper bounds checking when handling a failed HTTP request with specific content in the headers. By sending a specially crafted HTTP request, a remote attacker could overflow a buffer and execute arbitrary code on the system or cause a denial of service.
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.

Buffer Overflow

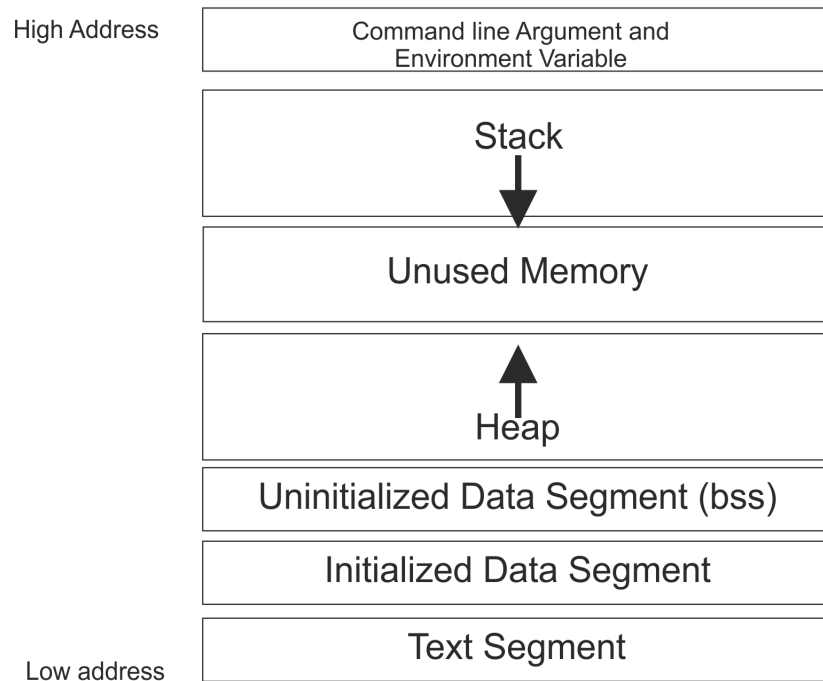
- O que acontece num *Buffer Overflow* accidental?
 - Programa torna-se instável;
 - Programa “crasha”;
 - Programa funciona de modo normal.
- Efeitos secundários dependem de
 - A quantidade de dados que é escrita após o fim do buffer;
 - Que dados (se algum) são sobrepostos/sobrescritos;
 - Se o programa tenta ler os dados sobrescritos;
 - Que dados substituem a memória que é sobrescrita.
- Fazer o debug de um problema deste género é normalmente difícil
 - Efeitos podem só aparecer muitas linhas de código depois.

Buffer Overflow

- Porque é que um *Buffer Overflow* pode ser um problema de segurança?
 - Pode ser explorado intencionalmente, e permitir que o atacante produza os efeitos que mais lhe convêm;
 - Objetivo é usualmente executar código com privilégios de administrador
 - O que é "simples" se o servidor estiver a executar com privilégios de administrador;
 - Ou pode ser feito após explorar o buffer overflow, através de outro ataque que permita o aumento de privilégios.
- Primeiro *paper* onde são descritos os ataques de *buffer overflow*: Aleph One, "*Smashing the Stack for Fun and Profit*", Phrack 49-14, 1996

Buffer overflow – organização da memória

- Organização da memória (RAM) na execução de um programa C



Text segment – segmento de código onde se encontram as instruções executáveis do programa (em código máquina/assembly).

Normalmente, este segmento é:

- partilhado, de modo a apenas estar uma cópia do código em memória,
- read-only, de modo a impedir a modificação accidental das instruções;

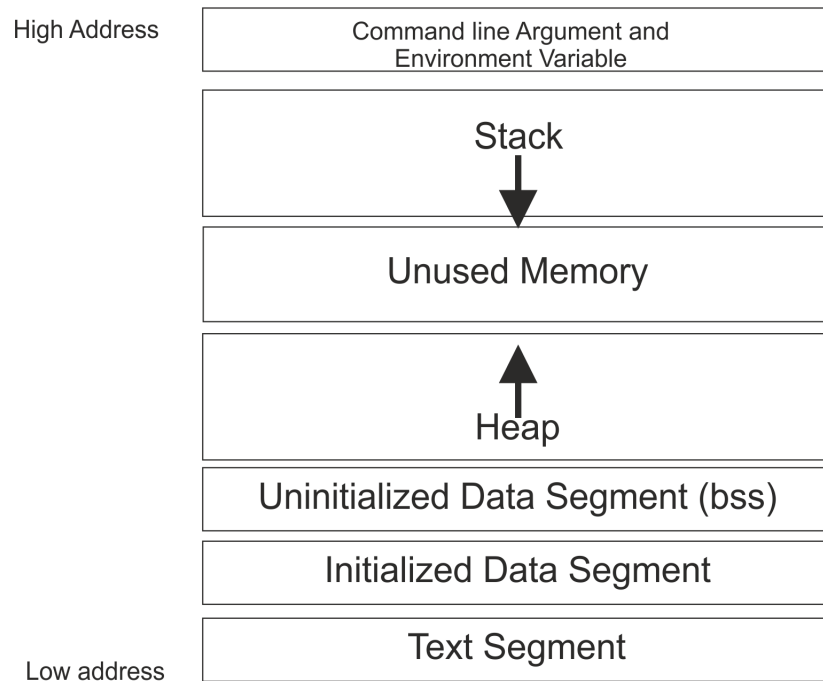
Initialized Data Segment – contém as variáveis globais e estáticas inicializadas pelo programador (e.g., `char s[] = "hello world"`, `int i = 10`);

Uninitialized Data Segment (BSS) – contém as variáveis globais e estáticas não inicializadas ou inicializadas a 0 (e.g., `int i`).

Nota: Todos estes segmentos (Texto, Dados e BSS) são conhecidos na altura da compilação.

Buffer overflow – organização da memória

- Organização da memória (RAM) na execução de um programa C



Stack – é uma estrutura LIFO que contém a stack do programa e cresce tipicamente (arquitetura x86 – PCs) dos endereços mais altos da memória para os mais baixos. Contém as variáveis das funções, assim como outra informação que é guardada de cada vez que uma função é chamada. O “stack pointer” aponta sempre para o topo da stack;

Heap – segmento para alocação dinâmica de memória (malloc, realloc, free). Este segmento é partilhado por todas as bibliotecas partilhadas e módulos dinâmicos de um processo;

Buffer overflow na Heap

Heap – segmento para alocação dinâmica de memória (malloc, realloc, free). Este segmento é partilhado por todas as bibliotecas partilhadas e módulos dinâmicos de um processo;

```
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);  
}
```

```
user@CSI:~/Aulas/Aula11.a/codigofonte$ ./a.out BOOMMMM...  
laranjas
```

Será que a variável *readonly* está fora do controlo do utilizador do programa?

Buffer overflow na Heap

```
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);
}
```

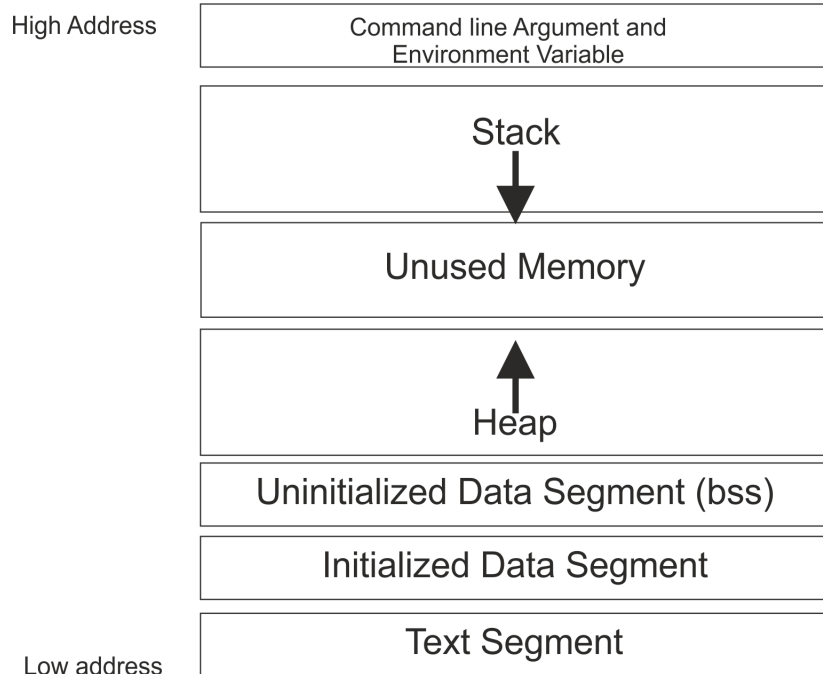
```
user@CSI:~$ ./a.out BOOMMMM...
Endereço da variavel dummy: 0x564dc9cf7010
Endereço da variavel readonly: 0x564dc9cf7030
laranjas
_
```

Buffer overflow na Heap

```
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 e funções (arquitetura x86/64 bits – PCs)

Função de origem:

1. Faz o push dos argumentos para a stack
2. Faz o push do endereço de retorno (i.e., endereço da instrução a executar imediatamente após lhe ser devolvido o controlo);
3. Salta para o endereço da função;

Função chamada:

1. Faz o push do apontador da frame (%rbp) antiga (relativa à stack da função de origem) para a stack;
2. O apontador da frame passa a ter o valor do apontador do topo da stack (%rbp = %rsp);
3. Faz o push das variáveis locais para a stack;

No retorno da função, a função chamada:

1. Obtém o apontador para a frame da função de origem (conteúdo do endereço %rbp);
2. Execução passa para o endereço de retorno, i.e., %rip = 8(%rbp)

Buffer overflow na Stack

```
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]);  
}
```

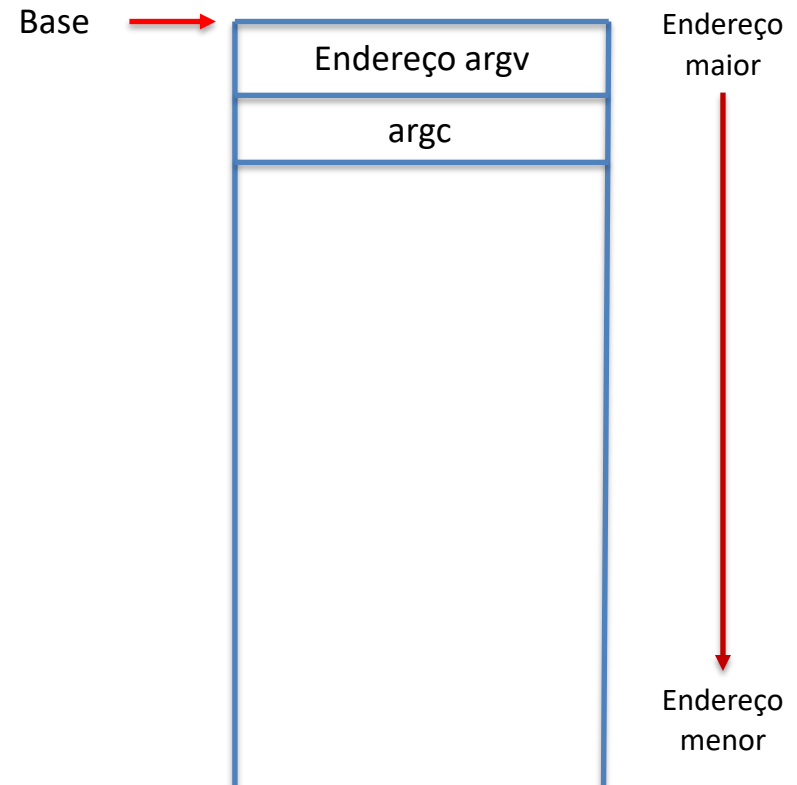
Stack – é uma estrutura LIFO que contém a stack do programa e cresce tipicamente (arquitetura x86 – PCs) dos endereços mais altos da memória para os mais baixos. Contém as variáveis das funções, assim como outra informação que é guardada de cada vez que uma função é chamada (incluindo o endereço de retorno da função). O “stack pointer” aponta sempre para o topo da stack.

Será que o utilizador do programa consegue chamar a função *debug()*?

Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

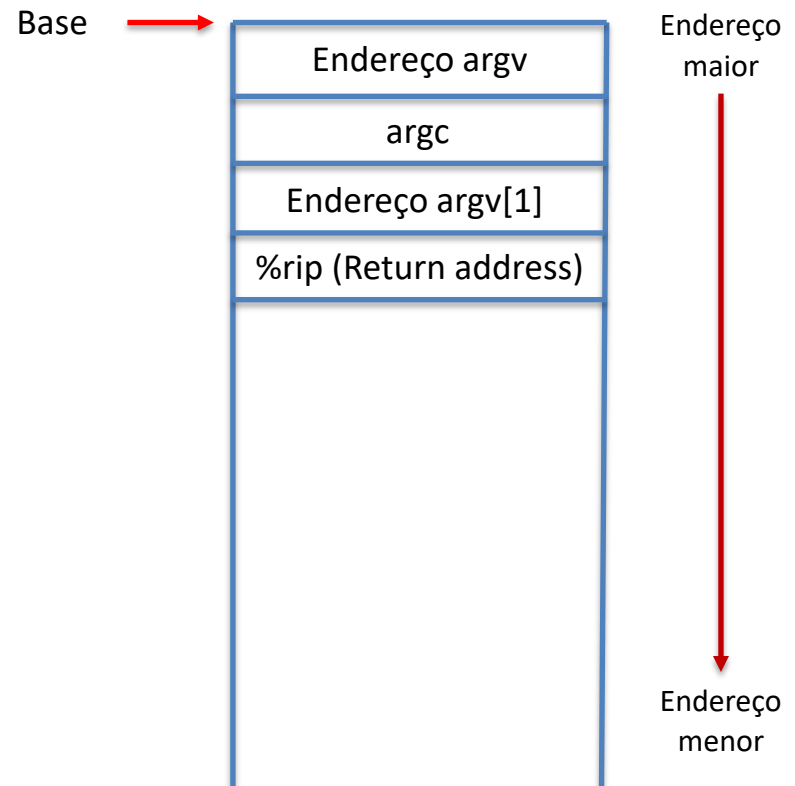
```
void debug() {  
    printf("Palavras-chave:\n");  
    printf("root: ola123\n");  
    printf("admin: 3eLdf75\n");  
}  
  
void store(char *valor) {  
    char buf[10];  
    strcpy(buf, valor);  
}  
  
→ int main(int argc, char **argv) {  
    store(argv[1]);  
}
```



Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
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]);  
}
```

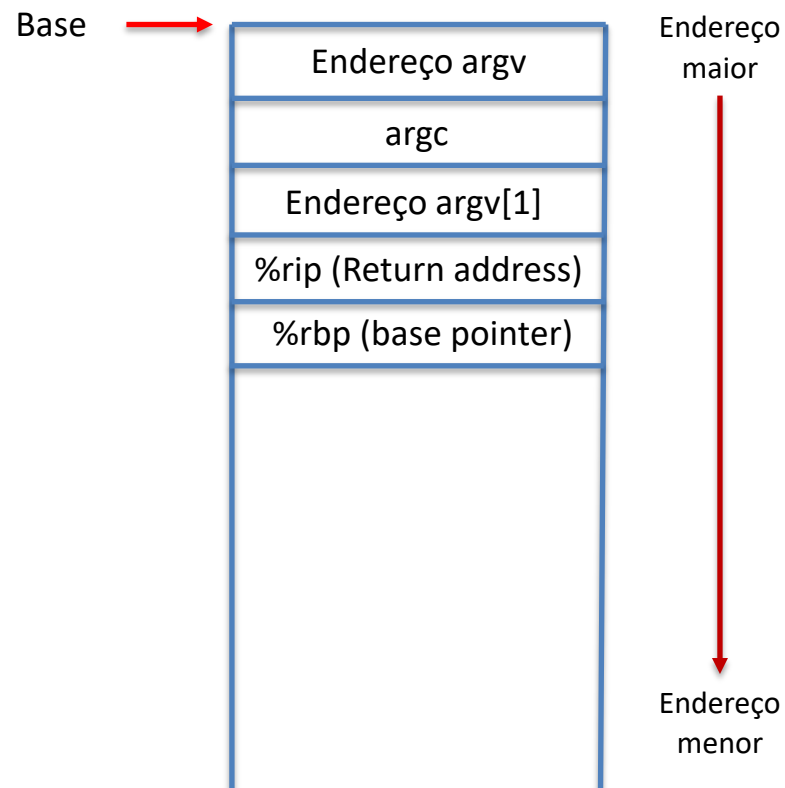


Nota: O “Return Address” também é designado por “%rip” (*instruction pointer*).

Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
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]);  
}
```



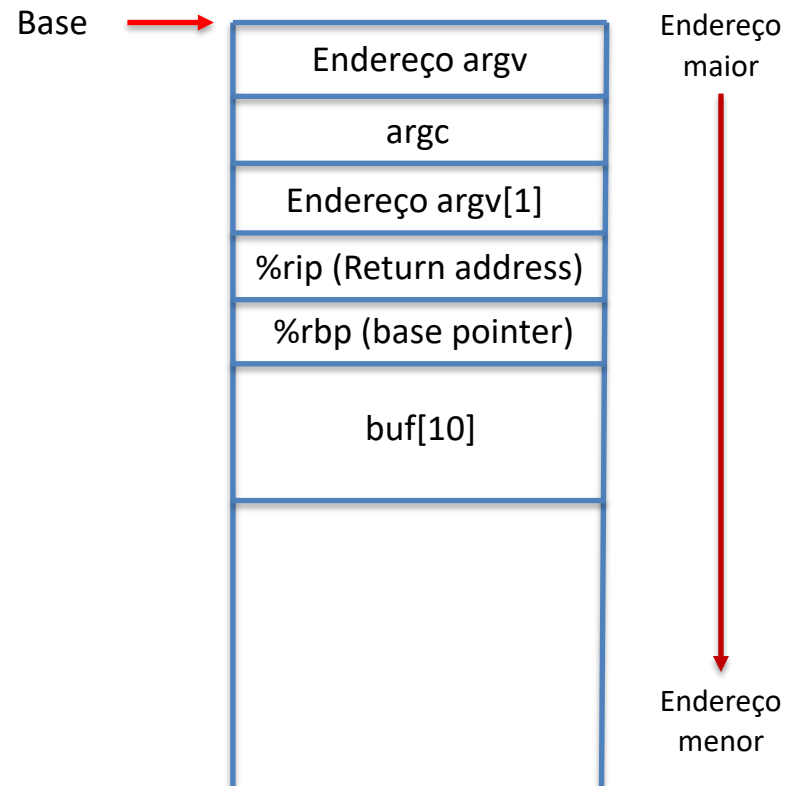
Buffer overflow na Stack

Vamos ver como se comporta a *stack*.

```
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]);
}
```



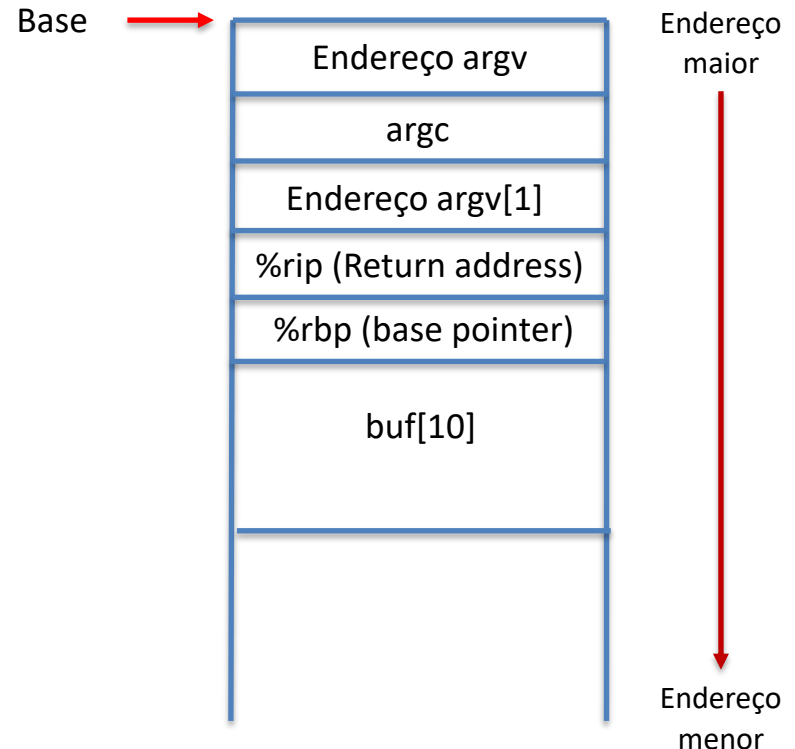
Antes de passar à função *strcpy*, a *stack* tem o aspeto indicado. O que fazer para alterar o programa, de modo a que o utilizador consiga chamar a função *debug()* ?

Buffer overflow na Stack

```
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]);
}
```



Antes da execução da função *strcpy*, a *stack* tem o aspeto indicado. O que fazer para alterar o programa, de modo a que o utilizador consiga chamar a função *debug()* ?

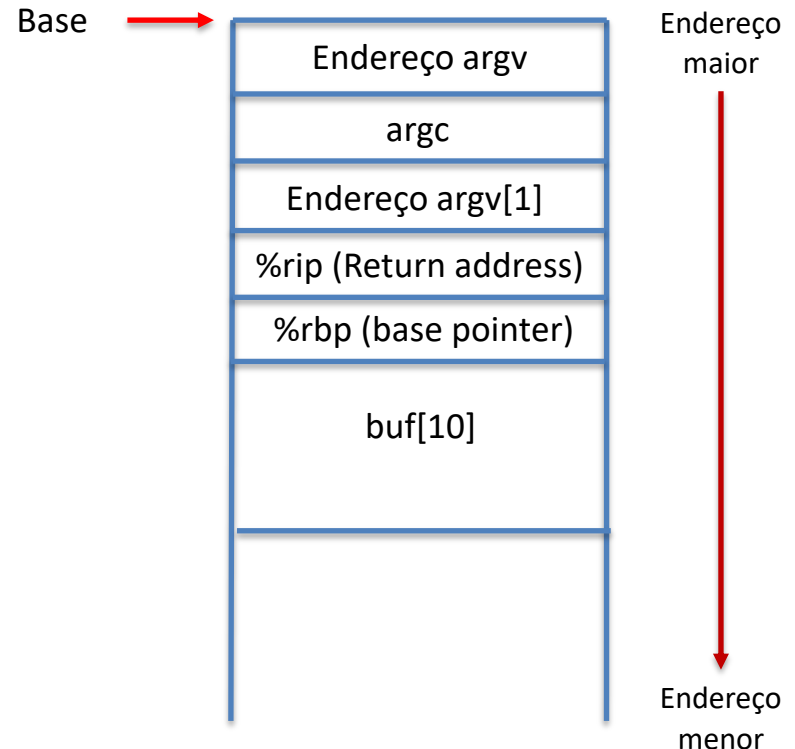
- Se o “Return address” apontar para a função *debug()*, a execução do programa será direcionada para a função *debug()*, no retorno da função *store()*;

Buffer overflow na Stack

```
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]);
}
```

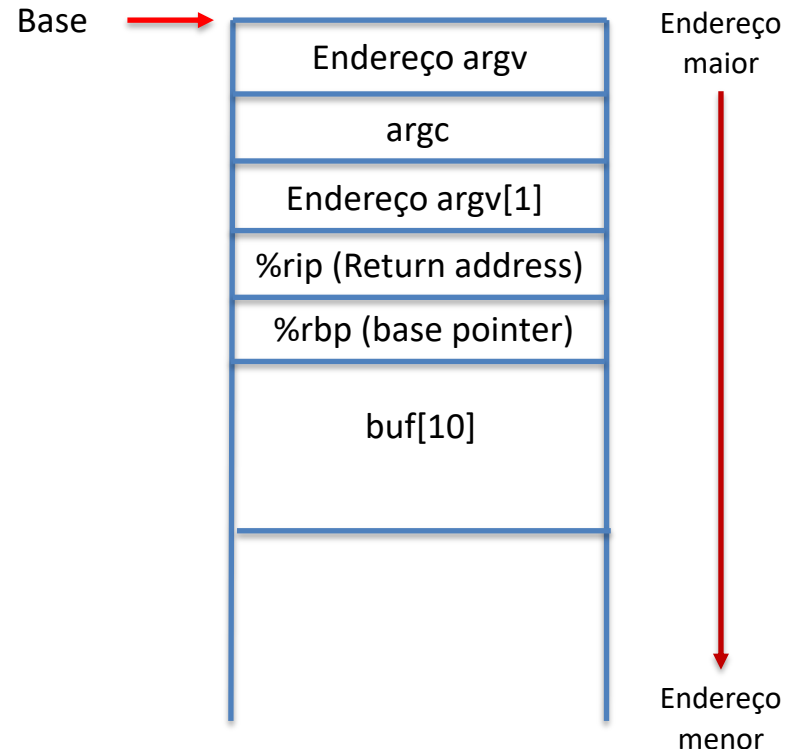


Antes da execução da função *strcpy*, a *stack* tem o aspeto indicado. O que fazer para alterar o programa, de modo a que o utilizador consiga chamar a função *debug()* ?

- Se o “Return address” apontar para a função *debug()*, a execução do programa será direcionada para a função *debug()*, no retorno da função *store()*;
- Para reescrever “Return address” temos que colocar em *buf* (10 + 8 + 8 bytes), sendo que os últimos 8 bytes devem corresponder ao endereço da função *debug()*.

Buffer overflow na Stack

```
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]);  
}
```



Antes da execução da função *strcpy*, a *stack* tem o aspeto indicado. O que fazer para alterar o programa, de modo a que o utilizador consiga chamar a função *debug()* ?

- Se o “Return address” apontar para a função *debug()*, a execução do programa será direcionada para a função *debug()*, no retorno da função *store()*;
- Para reescrever “Return address” temos que colocar em *buf* (10 + 8 + 8 bytes), sendo que os últimos 8 bytes devem corresponder ao endereço da função *debug()*.
- Vamos obter o endereço da função *debug()* alterando o programa para imprimir o seu endereço. (Na aula prática utilizaremos apenas um *debugger* para chegar ao mesmo resultado, sem alteração do programa.)

Buffer overflow na Stack

```
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
```

Temos todos os dados para conseguir executar a função debug().

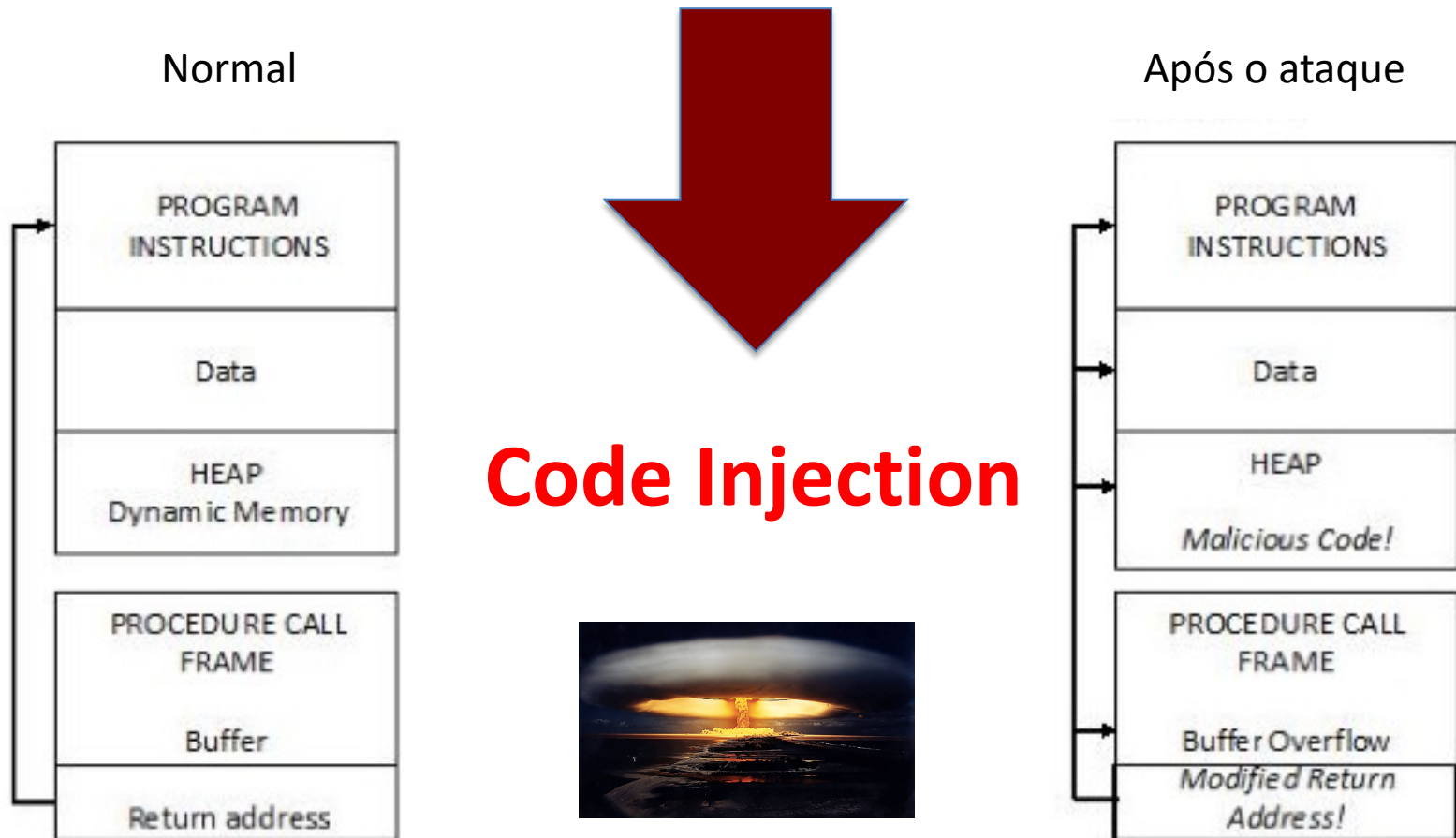
Nota: Lembre-se que o UNIX é um sistema little-endian, em que o byte menos significativo é colocado no endereço de memória mais baixo.

```
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

- Um fator importante nos ataques às vulnerabilidades de *buffer overflow*, foi o acesso ao código fonte.
 - Caso não exista, pode-se optar por uma estratégia de tentativa-erro (não é muito eficiente), ou então aplicar técnicas de engenharia reversa que permitam obter o código-fonte a partir do código binário.
- Outro fator determinante foi o endereço da função `debug()` não se alterar entre sucessivas execuções do programa.
 - Os compiladores mais recentes dos sistemas operativos principais já aleatorizam o espaço de endereçamento de uma aplicação e respectivas funções entre sucessivas execuções da mesma, pelo que tornam bastante improvável que o atacante consiga provocar a execução da função `debug()`.
 - Contudo, bastante improvável não significa impossível, como pode comprovar no livro *“Hacking – The art of exploitation (2nd edition), Jon Erickson”* (mas fora do âmbito desta disciplina).
- As consequências do ataque à vulnerabilidade do *buffer overflow* resumiram-se à execução de uma função existente no próprio programa atacado. No entanto, um atacante pretende executar código definido por ele próprio (por exemplo uma *shell*, em Linux) ou instalar uma ferramenta de administração remota (por exemplo, BackOrifice em Windows), o que é possível através de um ataque de *buffer overflow* na *stack* (fora do âmbito da disciplina), também denominado de *code injection*.

(Stack/Heap) Buffer overflow



Atacante, através de um (stack/heap) buffer overflow corrompe o endereço de retorno. Em vez de voltar para a função chamadora, o endereço de retorno devolve o controlo a código malicioso, localizado algures na memória do processo.

Read Overflow



- Exemplo: Heartbleed bug (<http://heartbleed.com/>)
 - SSL/TLS é o protocolo para comunicações cifradas na Web
 - Quando o URL começa por https, está-se a utilizar SSL/TLS
 - Heartbleed é um bug existente na implementação OpenSSL (uma das mais utilizadas) – versões 1.0.1 a 1.0.1f – do SSL/TLS
 - Bug descoberto em Março 2014, estava na versão disponibilizada desde Março 2012!!!
 - O servidor SSL deve aceitar uma mensagem “heartbeat” que ecoa de volta;
 - A mensagem de “heartbeat” indica o tamanho do eco a devolver, mas o servidor SSL não validava o tamanho;
 - Desse modo, o atacante podia pedir um tamanho maior e ler para além do conteúdo da mensagem, o que permitia o acesso à memória do servidor e o acesso aos dados confidenciais (passwords, chaves, informação de ID, ...) protegidos pelo SSL/TLS;
 - O ataque não deixa rasto !!!

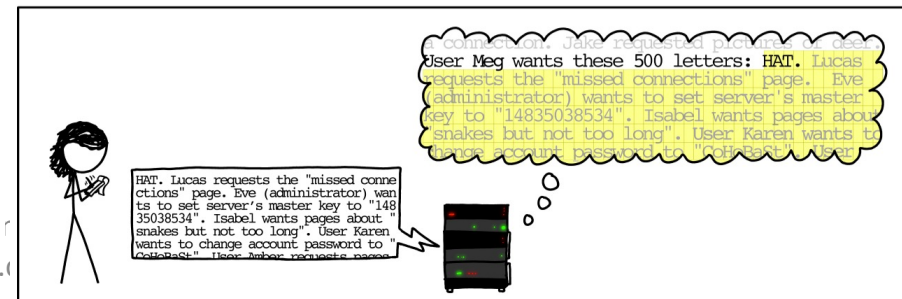
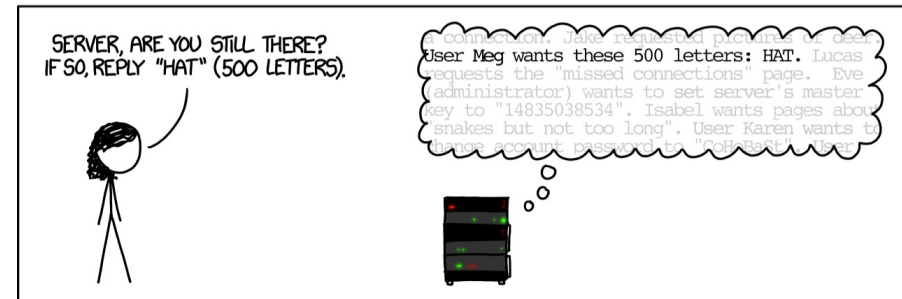
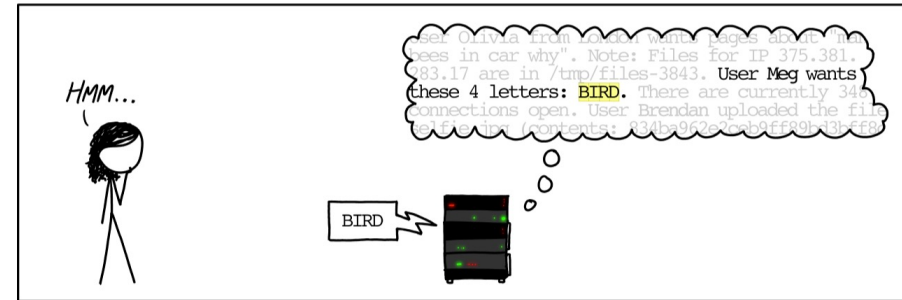
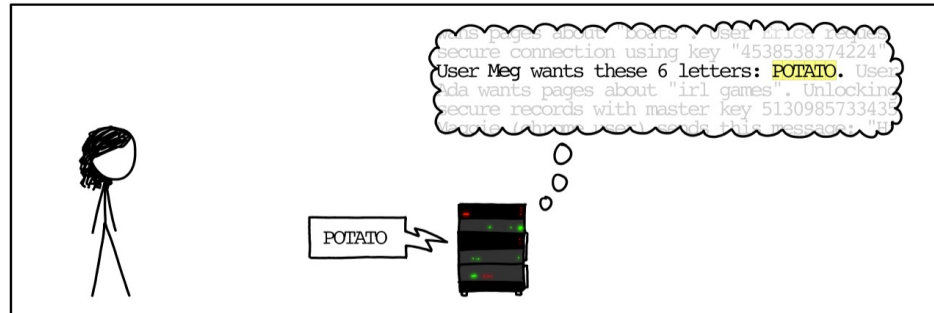
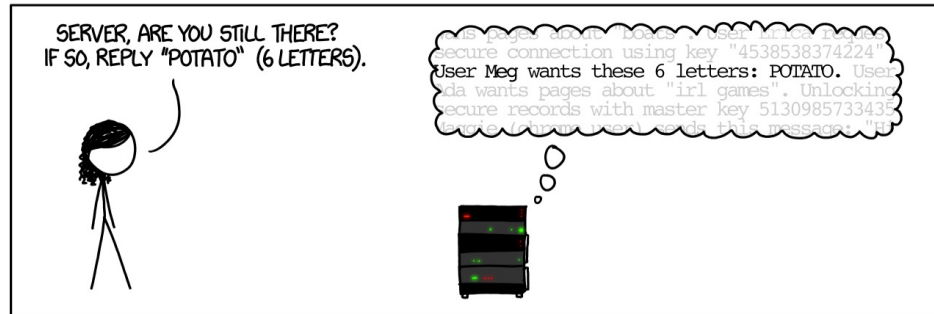


Read Overflow

- Exemplo: Heartbleed bug (<https://xkcd.com/1354/>)



HOW THE HEARTBLEED BUG WORKS:



Reduzir vulnerabilidades de *Buffer overflow*

- Programação defensiva:
 - Validar índices: Verificando se os valores dos índices são inteiros e estão dentro dos limites de endereçamento do array. Esta validação é mandatória para valores fornecidos pelo utilizador ou por outra fonte de input não confiável (e.g., informação lida de um ficheiro ou obtida através de uma conexão de rede).
 - Atenção aos ciclos (for, while, ...) !
 - Atenção aos métodos que possam modificar os índices de um array !
 - Espaço alocado: Antes de copiar os dados, garantir que a variável de destino tem espaço suficiente para guardar esses dados. Se não tiver espaço suficiente, não copiar mais dados do que o espaço disponível.
 - Tamanho do array: as linguagens de programação têm funções que devolvem o tamanho alocado para um array. Utilize-as!
 - Se utilizar um array como argumento de uma função, utilize outro argumento para enviar também o tamanho do array. Esse valor pode ser utilizado como limite máximo do índice do array.
 - Estruturas de dados alternativas: vulnerabilidades de *buffer overflow* podem ser reduzidas se utilizar estruturas de dados alternativas, como vectores e iteradores. Utilize-os !
 - Alocar memória: Sempre que possível, aloque memória só após saber quanta necessita.
 - Evite funções de risco: Ao utilizar funções para ler, copiar dados ou alocar/libertar memória, utilize bibliotecas que forneçam versões mais seguras que as funções standard;
 - Utilize as ferramentas: Avisos de compiladores no caso de potenciais buffer overflows. Ferramentas de análise estática para analisar o código fonte ou de análise dinâmica para examinar o estado do programa em execução.
 - Recuperação: Se o programa não poder continuar, tem que se garantir uma recuperação adequada. Nota: Trate as excepções com cuidado!