Buffer Overflow

- Vulnerabilidade que permite a um atacante injetar código num programa em execução
  - O programa executa depois o código injetado.
  - O atacante pode ganhar o controlo da máquina.

Buffer Overflow

- *Buffer*: memória para armazenar dados introduzidos pelo utilizador
- *Buffer Overflow*: Quando os dados introduzidos excedem a capacidade do buffer
  - Os caracteres extra vão para zonas de memória para onde era suposto não irem
- Podem acontecer em linguagens que não verificam se o limite dos *buffers* são excedidos
  - Por exemplo, C e C++

Buffer Overflow: Exemplo

```
#define BUFFSIZE 1024
(...)
char buff[BUFFSIZE];
gets(buff);
```

- **`buff`** tem capacidade para 1024 caracteres
- Utilizador introduz mais de 1024 caracteres
- Caracteres excedentes saem fora do buffer (*buffer overflow*)

Buffer Overflows: Outro exemplo

```
function foo(char *a){
        char b[100];
        …
        strcpy(b,a);//(dest,source)
        …
```

- Qual o tamanho do array de caracteres apontado por “a”?
- Será que a o *array* de caracteres apontada por ”a” é uma *string*, i.e., termina com o caracter nulo?

Buffer Overflow: Efeitos

- Os efeitos de um *buffer overflow* dependem:
  - Da quantidade de dados escritos para além do fim do buffer
  - Do tipo de dados que estavam na memória que e que foram sobrepostos
  - Dos novos dados que são colocados na memória, sobrepondo-se aos que lá estavam
  - Das ações que o programa fará com os novos dados colocados na memória
- Possíveis efeitos:
  - *Crash*
  - Instabilidade
  - Continuar funcionamento normal

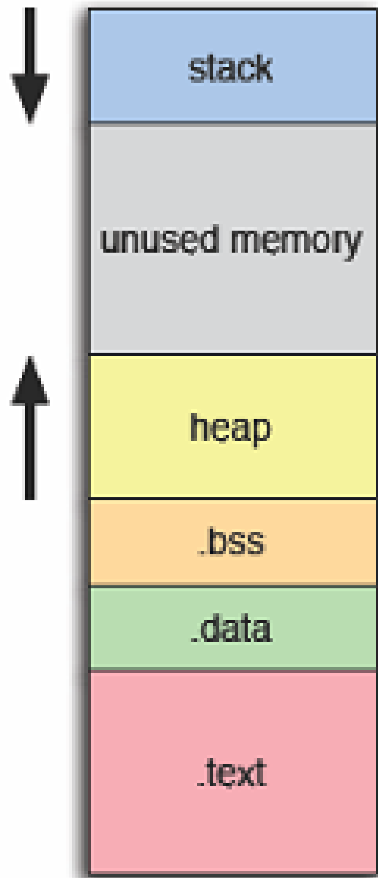
Buffer Overflows: Problema sério

- Podem ser explorados para modificar:
  - Variáveis
  - Ponteiros para dados
  - Ponteiros para funções
  - Endereço de retorno na stack
- Pode permitir ao atacante executar o seu próprio código. Muito Grave!

Tipos de *Buffer Overflows*

- *Overflow* na pilha (*stack*)
  - Quando o *buffer* vulnerável está armazenado na pilha
    - Vai ser detalhado a seguir.
  - *Smashing the stack*
- *Overflow* na *heap* (monte)
  - Quando o *buffer* vulnerável é armazenado na *heap*
    - *Heap*: zona de memória onde são guardados dados dinâmicos.
    - Ex: memória reservada com as funções malloc

Linux x86 Process Memory Layout

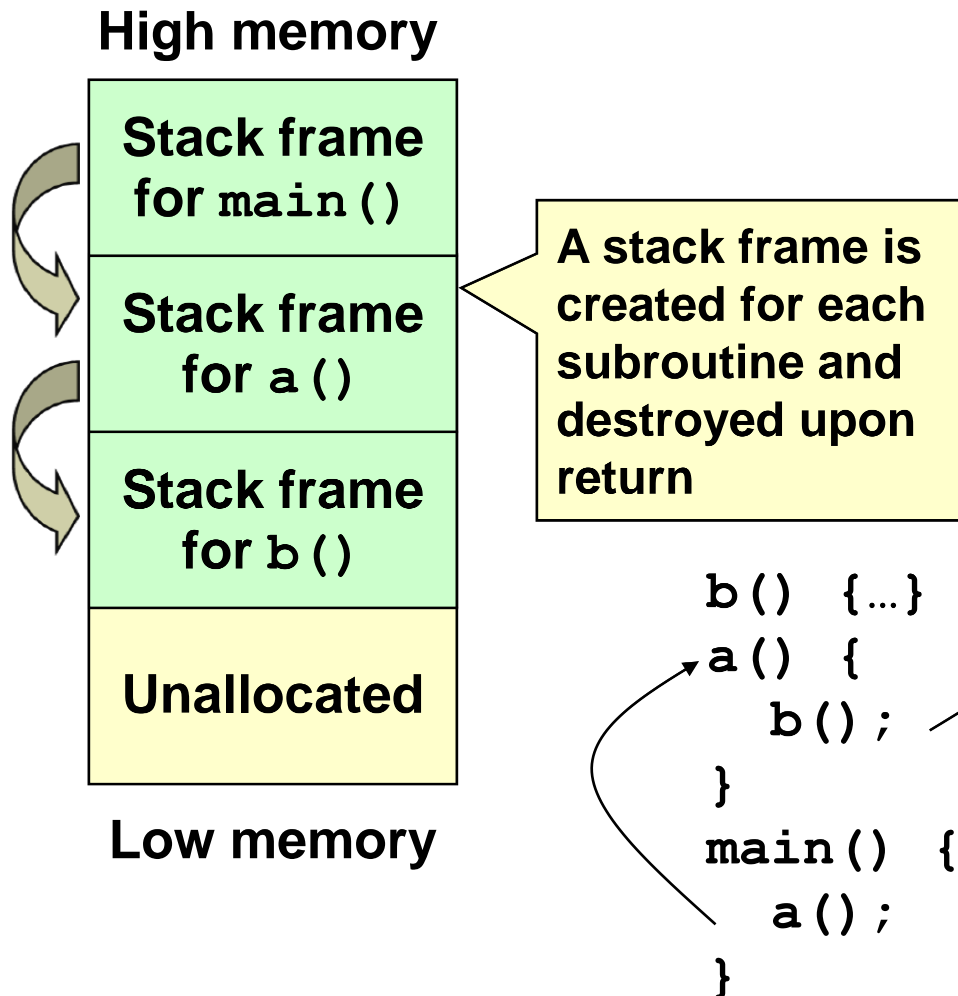


- Memória do processo está particionada em segmentos
  - .text: Código do programa
  - .data: variáveis estáticas iniciadas
  - .bss: variáveis estáticas não iniciadas
  - heap: memória alocada dinamicamente
  - stack: pilha de chamadas do programa

Pilha (*stack*)

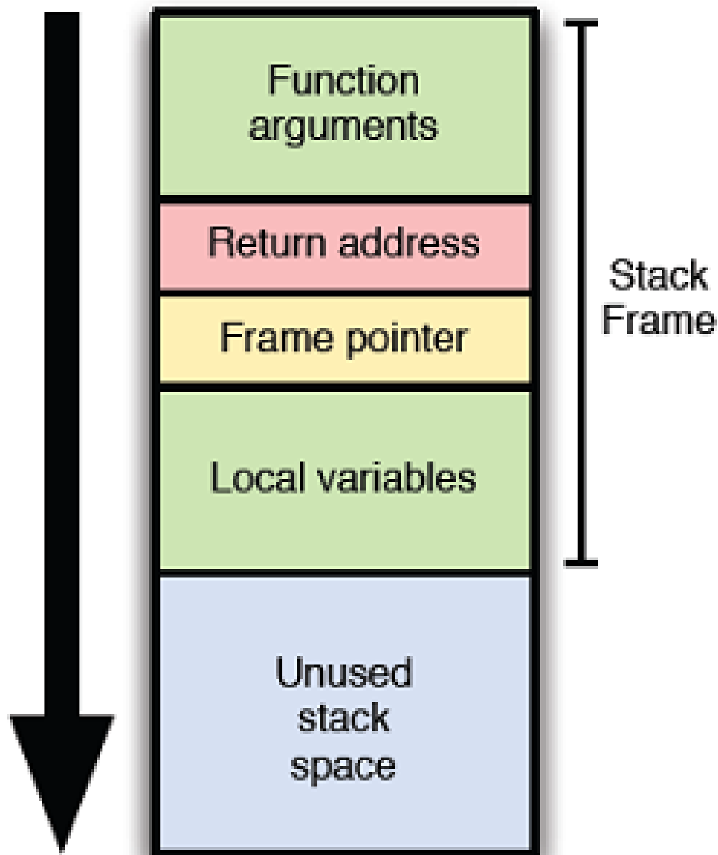
- Normalmente cresce na direcção dos endereços mais baixos
  - Arquitecturas Intel, Motorola, SPARC e MIPS
- O *stack pointer* (SP) aponta para o topo da pilha
  - Último endereço da pilha
  - Registo "esp" na arquitectura Intel

Organização da Pilha em Quadros (*Frame Structure*)



- A pilha (*stack*) é composta por quadros (*frames*)
- Quadros são colocados na pilha como consequência da chamada a funções
  - prólogo da função (*function prolog*)
- O endereço do quadro actual está armazenado no registo *Frame Pointer* (FP)
  - registo "ebp" na arquitectura Intel

Estrutura de um Quadro da Pilha (*Stack Frame*)



- Cada quadro da pilha contém:
  - Os parâmetros de entrada para a função actual
  - O endereço de retorno, para onde o programa salta após a execução da função
  - O ponteiro para o quadro anterior
  - Variáveis locais da função
- Os quadros crescem desde os endereços de memória mais altos para os mais baixos.

Buffer Overflow na Pilha: Exemplo (1)

- Considere o seguinte exemplo:

```
void write_str(char *str)
{
        char buffer[8];

        strcpy(buffer, str);
        printf("Buffer: %s\n", buffer);
        return;
}
int main(int argc, char *argv[])
{
        write_str( argv[1] );
        return 0;
}
```

Buffer Overflow na Pilha: Exemplo (2)

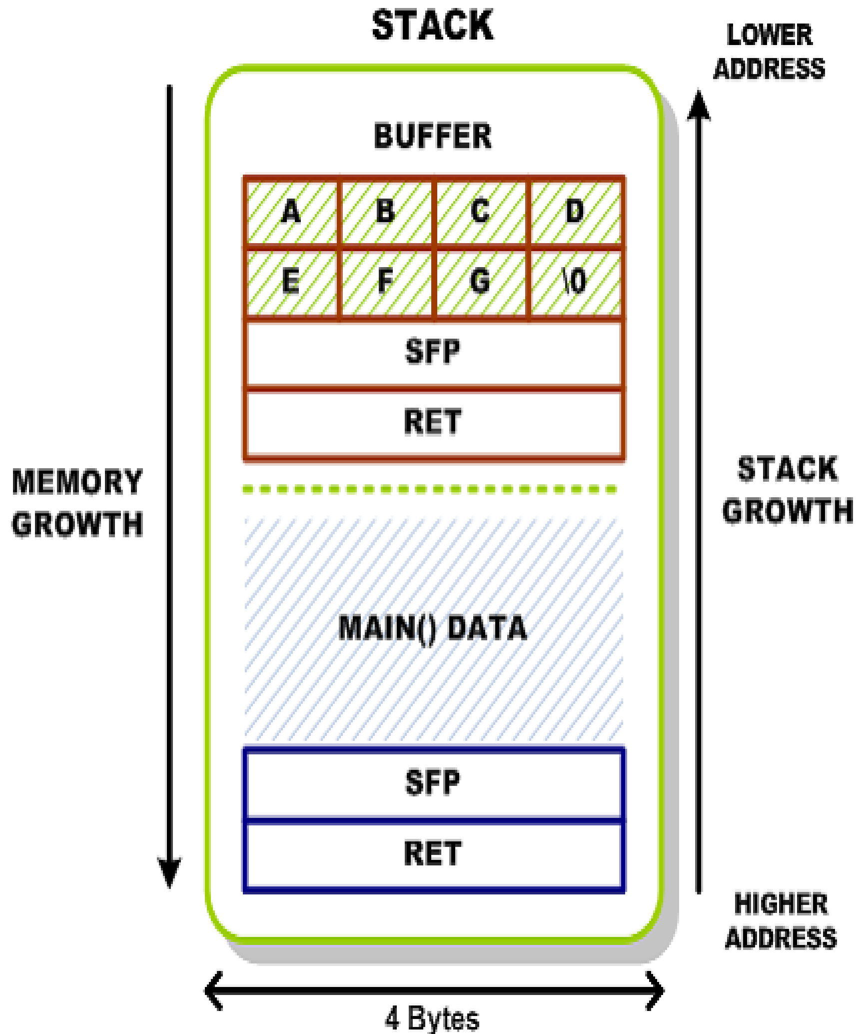
- Resultados da execução do programa com diferentes entradas:

```
$ ./example1 ABCDEFG
buffer: ABCDEFG

$ ./example1 AAAAAAAAAAAAAAAAAAA
buffer: AAAAAAAAAAAAAAAAAAA
Segmentation fault
```

- No segundo caso gerou a mensagem *Segmentation Fault*?! Porquê?

Buffer Overflow na Pilha: Exemplo (3)



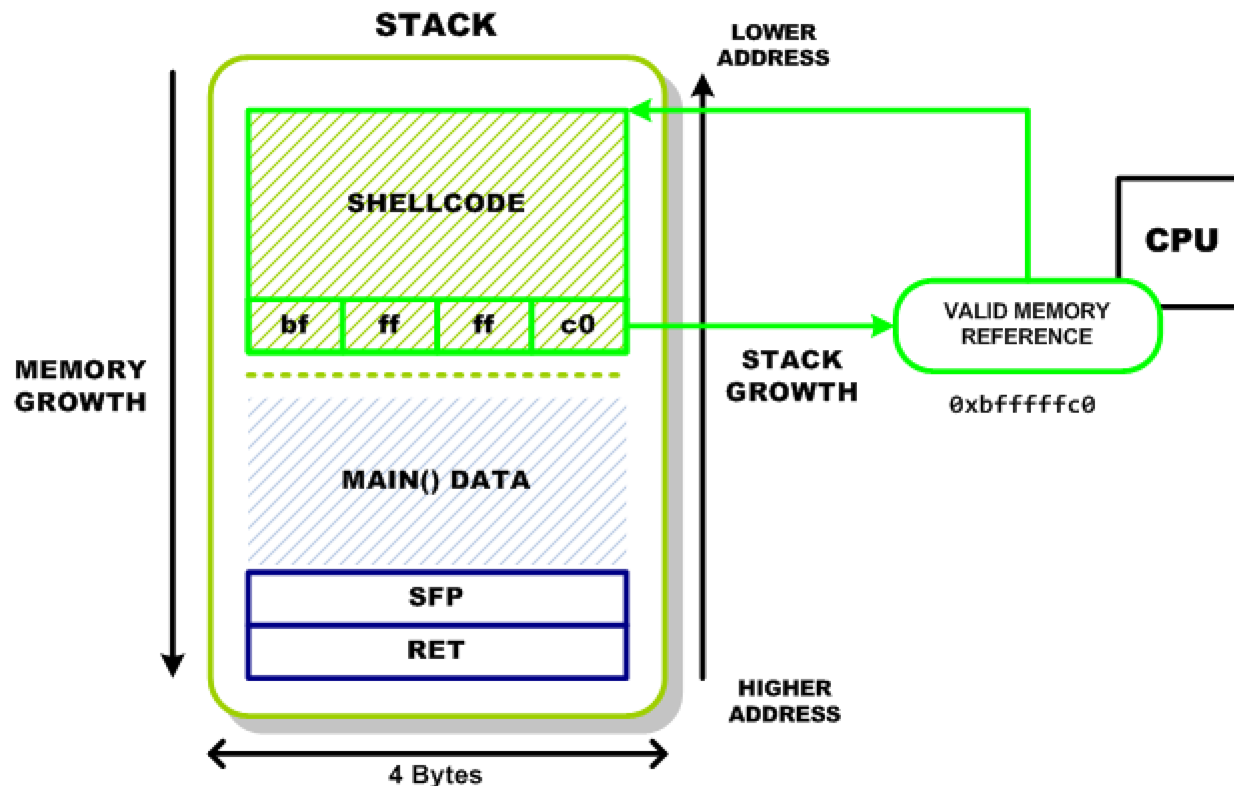
- Estado da stack:
  - Antes de sair da função write_str(char *str)
  - Com o parâmetro de entrada “ABCDEFG”

Buffer Overflow na Pilha: Exemplo (4)

- Estado da stack:
  - Antes de sair da função write_str(char *str)
  - Com o parâmetro de entrada “AAAAAAAAAAAAAAAAAAAA”

Buffer Overflow: Exploração

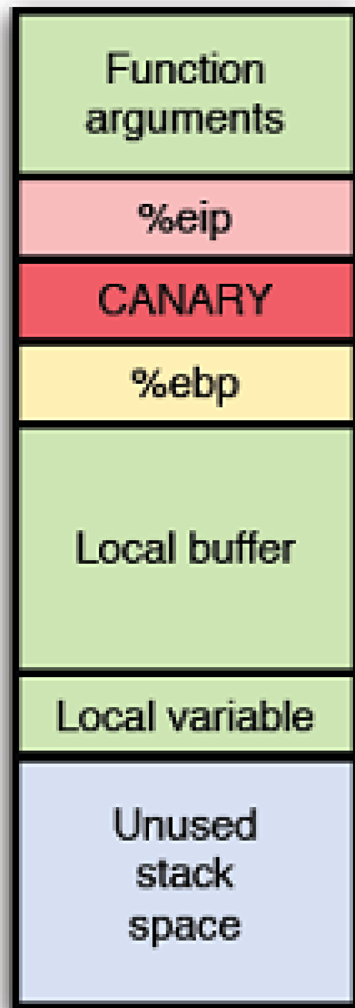
- O que acontece se o endereço de retorno for sobreposto com um endereço válido?



Soluções contra *Buffer Overflows*

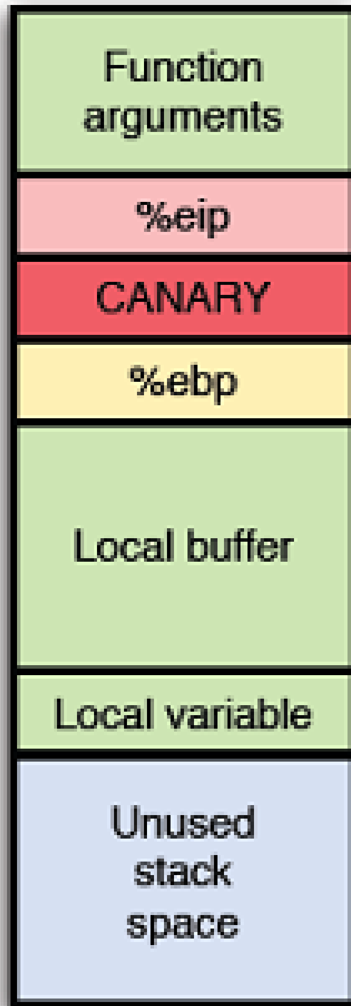
- Programação
  - Escrever programas decentes
    - Programação defensiva
  - Utilização de linguagens que verificam limites (*bound checking*):
    - Java, C#, Pyton
  - Utilizando funções que façam a verificação de limites
    - Em C, por exemplo, utilizar fgets() em vez de gets()
- Sistemas Operativos e Compiladores
  - Utilização de canários
  - Address Space Layout Randomization (ASLR)
  - Memória não executável
- "There are no such things as dangerous functions, only dangerous developers", Dave Cutler, responsável pelo desenv. Windows NT

Canários



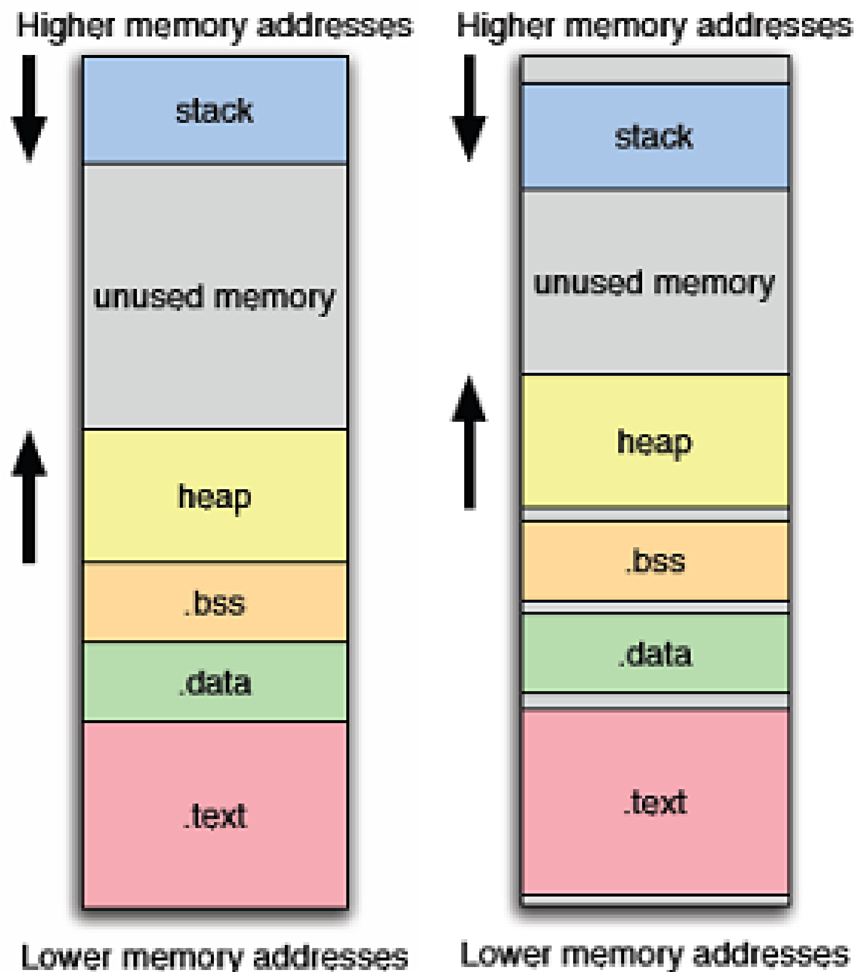
- Técnica para detectar e prevenir *buffer overflows* colocando um canário à frente da informação sensível
  - Canário: conjunto de alguns bytes com valores especiais.
- Se o canário foi destruído, então ocorreu um *buffer overflow*
- Existem canários tanto para a pilha como para a *heap*

Canários na pilha



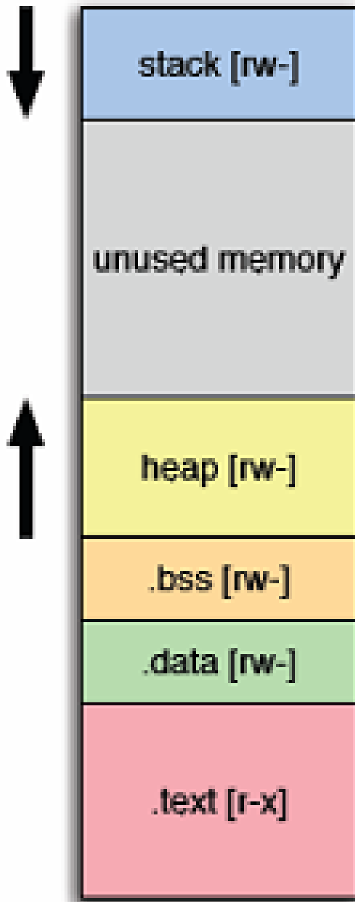
- Canário para protecção do endereço de retorno
  - O canário deve ser colocado entre o endereço de retorno e as variáveis na stack.
- Antes de retornar, o valor do canário deve ser verificado
  - Se não está intacto
    - Houve um buffer overflow
    - O programa deve terminar
- Se o objectivo do ataque era a negação do serviço, então o ataque teve sucesso
  - Mas a máquina não foi comprometida

Address Space Layout Randomization (ASLR)



- Técnica para de forma aleatória perturbar a localização das áreas de memória
- Força o atacante a adivinhar os endereços importantes de código e dados
  - Baixa probabilidade
- Eficácia dependente da quantidade de entropia introduzida

Memória não-executável



- Técnica para alocar memória de forma exclusiva para código ou para dados
- Pode ser implementado por hardware, sob a forma de bits de escrita PTE (Page Table Entry), ou emulado por software
- Impede um atacante de injectar dados para serem executados como código