

# Fundamentos de Segurança Informática (FSI)

2021/2022 - LEIC

**Manuel Barbosa**  
**mbb@fc.up.pt**

# **Aula 3**

## **Controlo (Parte 2)**

# Buffer Overflows na Heap

# Overflows na Heap

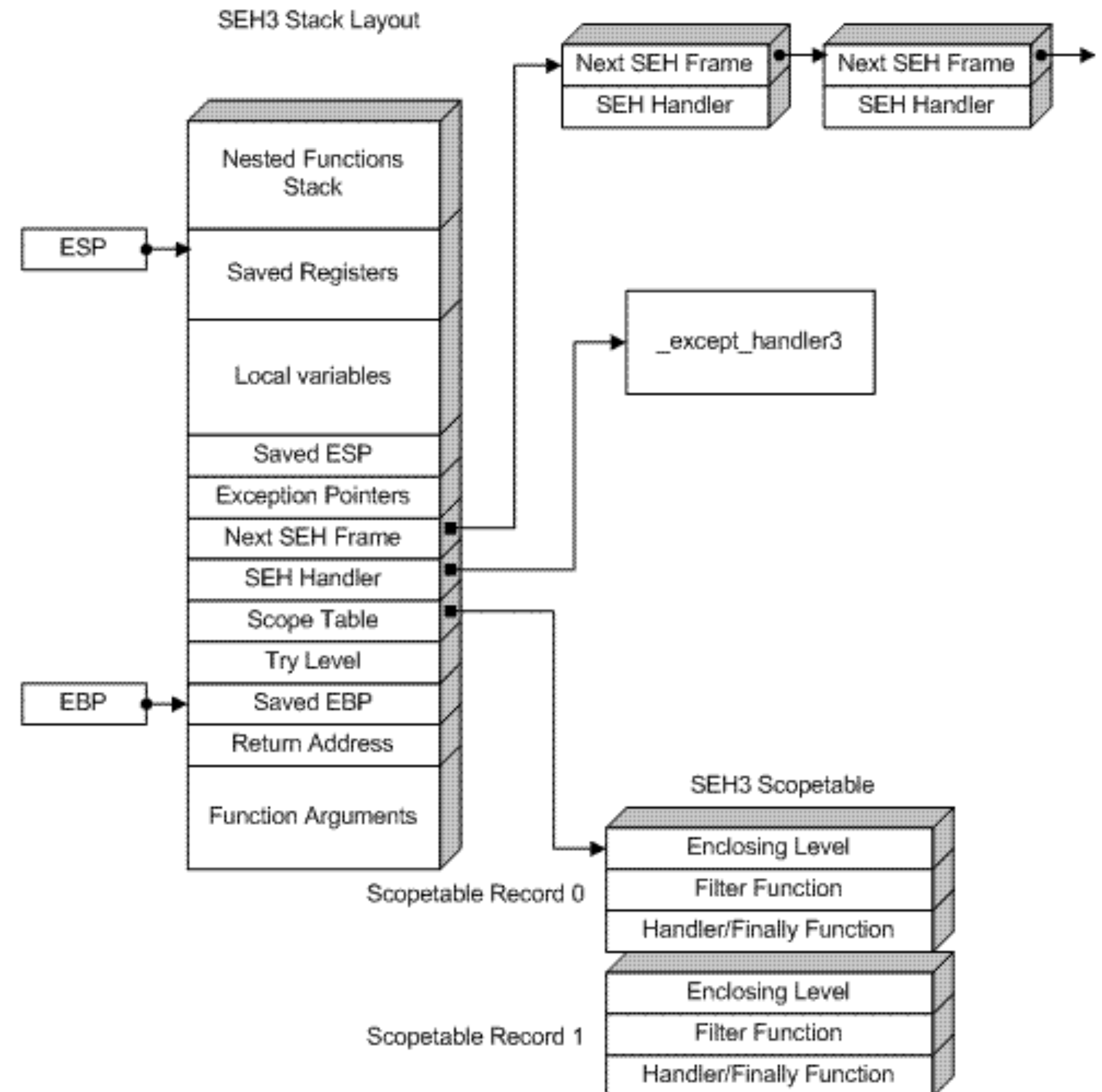
- Quando ocorre um buffer overflow na heap a obtenção de controlo também é possível.
- Existem muitos candidatos para alteração que permitem obter controlo:
  - apontadores para funções  
(virtual function tables em C++ e global offset tables para bibliotecas com dynamic linking)
  - exception handlers  
(lista ligada de apontadores para funções)
  - longjmp buffers para tratamento de exceções em C  
(mais apontadores para funções utilizados)

The **setjmp()** function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer *env* for later use by **longjmp()**. In this case, **setjmp()** returns 0.

The **longjmp()** function uses the information saved in *env* to transfer control back to the point where **setjmp()** was called and to restore ("rewind") the stack to its state at the time of the **setjmp()** call. In addition, and depending on the implementation (see NOTES), the values of some other registers and the process signal mask may be restored to their state at the time of the **setjmp()** call.

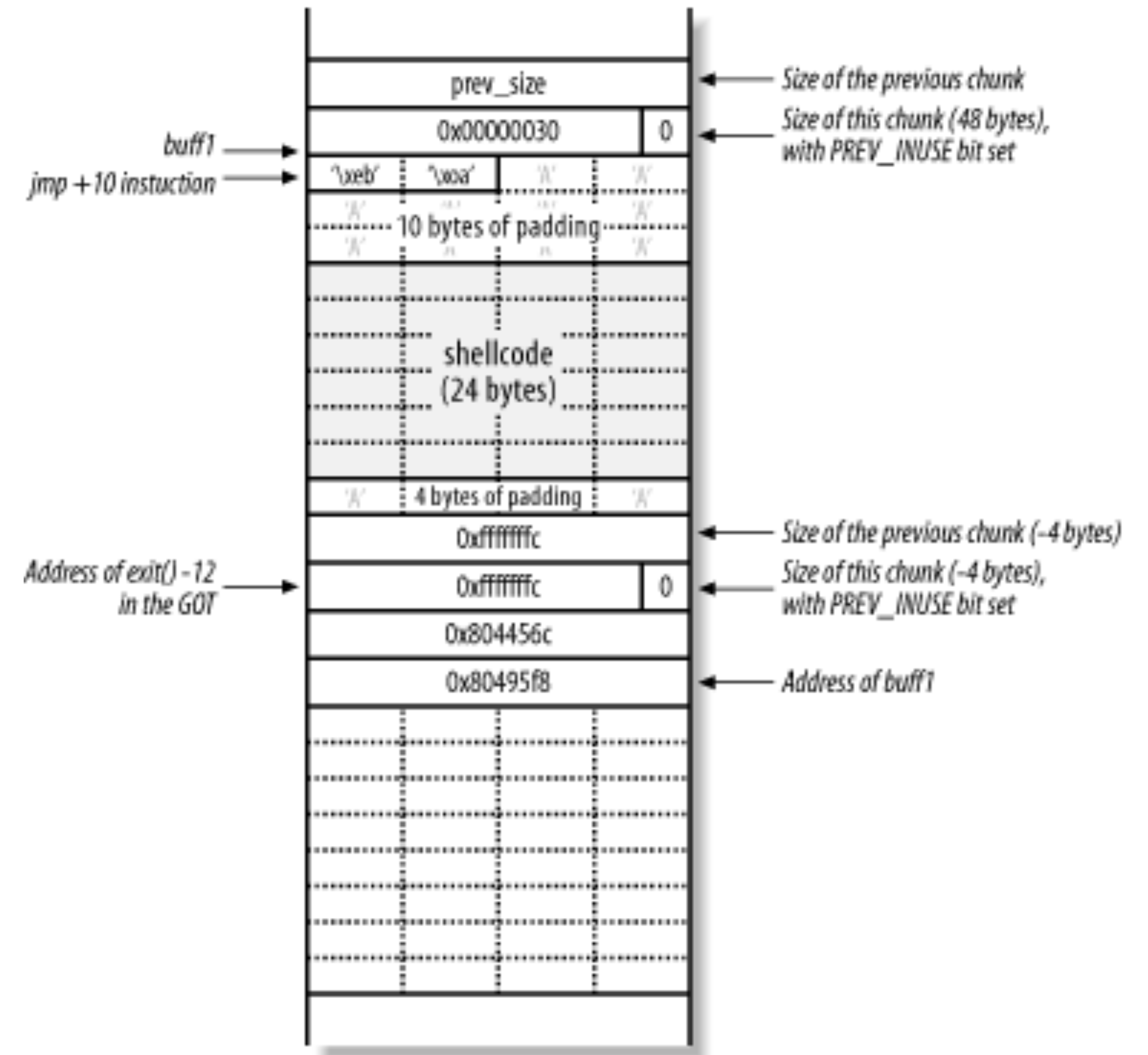
# Tratamento de Exceções

- Em linguagens com tratamento de exceções (e.g., C++):
  - A stack frame de uma função inclui uma lista ligada/tabela de apontadores para as diferentes rotinas de tratamento de exceções
  - Se existe um buffer overflow na heap, pode escrever-se por cima de um desses endereços
  - Se a exceção ocorrer, a execução prosseguirá para o endereço da nossa escolha.



# Overflows na Heap

- Uma generalização chamada "smashing the heap" funciona da seguinte forma:
  - para além de colocar a "payload" em memória, reescreve-se informação de gestão utilizada por malloc/free
- as operações de malloc/free seguintes são utilizadas para alterar outras partes da memória:
  - o free apaga um elemento de uma lista duplamente ligada reescrevendo apontadores
  - manipulado para escrever numa parte da memória completamente diferente
  - de facto reescreve um apontador para uma função com o endereço de código malicioso





# Exemplos de Ataques por Heap Overflows

## Overview

Microsoft Windows Media Player contains a buffer overflow vulnerability that may allow a remote, unauthenticated attacker to execute arbitrary code on a vulnerable system.

## Description

Microsoft Windows Media Player (WMP) is an application that ships with Microsoft Windows systems used to play various types of media files. Windows Media Player fails to properly validate bitmap image files (.bmp), potentially allowing a buffer overflow to occur.

If the size field in the bitmap header is set to set to a value that is less than the actual size of the file, WMP will allocate an under-sized buffer to hold the bitmap. When data is copied to this buffer, the buffer overflow may occur. For more information, please see Microsoft Security Bulletin [MS06-005](#).

Windows Media Player can play bit map format files, such as a .bmp file and use Windows Media Player (WMP) to decode the .dll process bmp file. But it can't correctly process a bmp file which declares it's size as 0. In this case, WMP will allocate a heap size of 0 but in fact, it will copy to the heap with the real file length. So a special bmp file that declares it's size as 0 will cause the overflow. When changing the size

Buffer overflow in the exif\_read\_data function in PHP before 4.3.10 and PHP 5.x up to 5.0.2 allows remote attackers to execute arbitrary code via a long section name in an image file.

The bug was discovered 12/15/2004. The weakness was presented 01/10/2005 (Website). The advisory is shared at [redhat.com](#). This vulnerability is uniquely identified as [CVE-2004-1065](#) since 11/23/2004. The exploitability is told to be easy. It is possible to initiate the attack remotely. No form of authentication is needed for exploitation. Technical details are known, but no exploit is available. The price for an exploit might be around USD \$0-\$5k at the moment ([estimation calculated on 06/05/2019](#)).

The vulnerability was handled as a non-public zero-day exploit for at least 2 days. During that time the estimated underground price was around \$25k-\$100k. 005056 running

## ‘Multiple Vulnerabilities within PHP 4/5 (pack, unpack, safe\_mode\_exec\_dir, safe\_mode, realpath, unserialize)’

Published on December 16th, 2004

[06 - unserialize() - wrong handling of negative references ]

The variable `unserializer` could be fooled with negative references to add false `zvalues` to `hashtables`. When those `hashtables` get destroyed this can lead to `efree()`s of arbitrary memory addresses which can result in arbitrary code execution. (Unless Hardened-PHP's memory manager canaries are activated)

[07 - unserialize() - wrong handling of references to freed data ]

Additionally to bug 07 the previous version of the variable `unserializer` allowed setting references to already freed entries in the variable `hash`. A skilled attacker can exploit this to create an universal string that will pass execution to an arbitrary memory address when it is passed to `unserialize()`. For AMD64 systems a string was developed that directly passes execution to code contained in the string itself.

ties (d47e9d19-5016-11d9-9b5f-eBSD Local Security Checks and

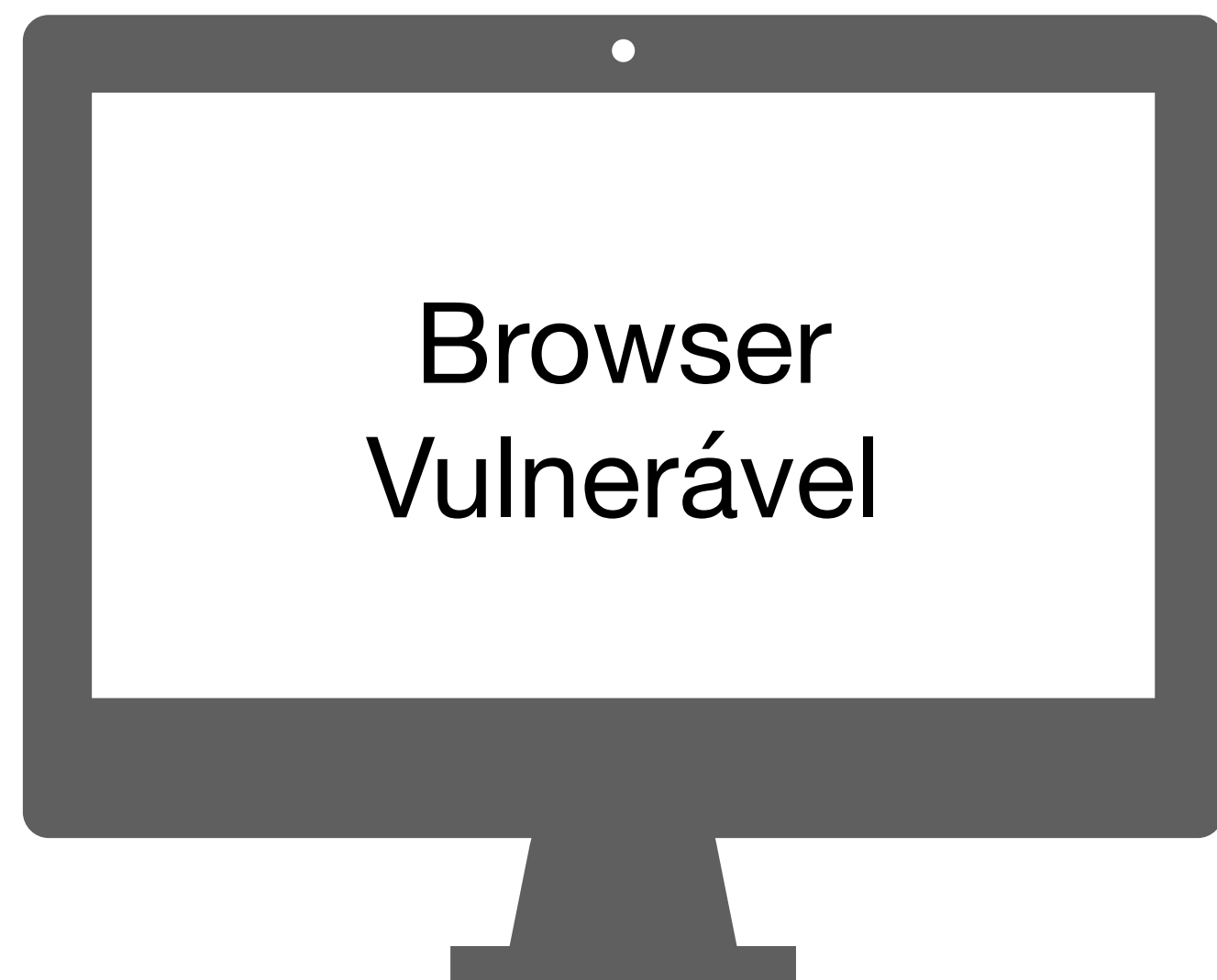
# Criação do Exploit

- Domínio de programação baixo nível (assembly) e debugging de código binário
- Compreender as causas do overflow e como o desencadear de forma controlada
- Replicar das condições de execução do código alvo:
  - prever endereços a alterar e
  - prever localização de shellcode
  - evitar crash antes de tomada de controlo
- E quando isto não é possível? (veremos mais tarde diversas razões)

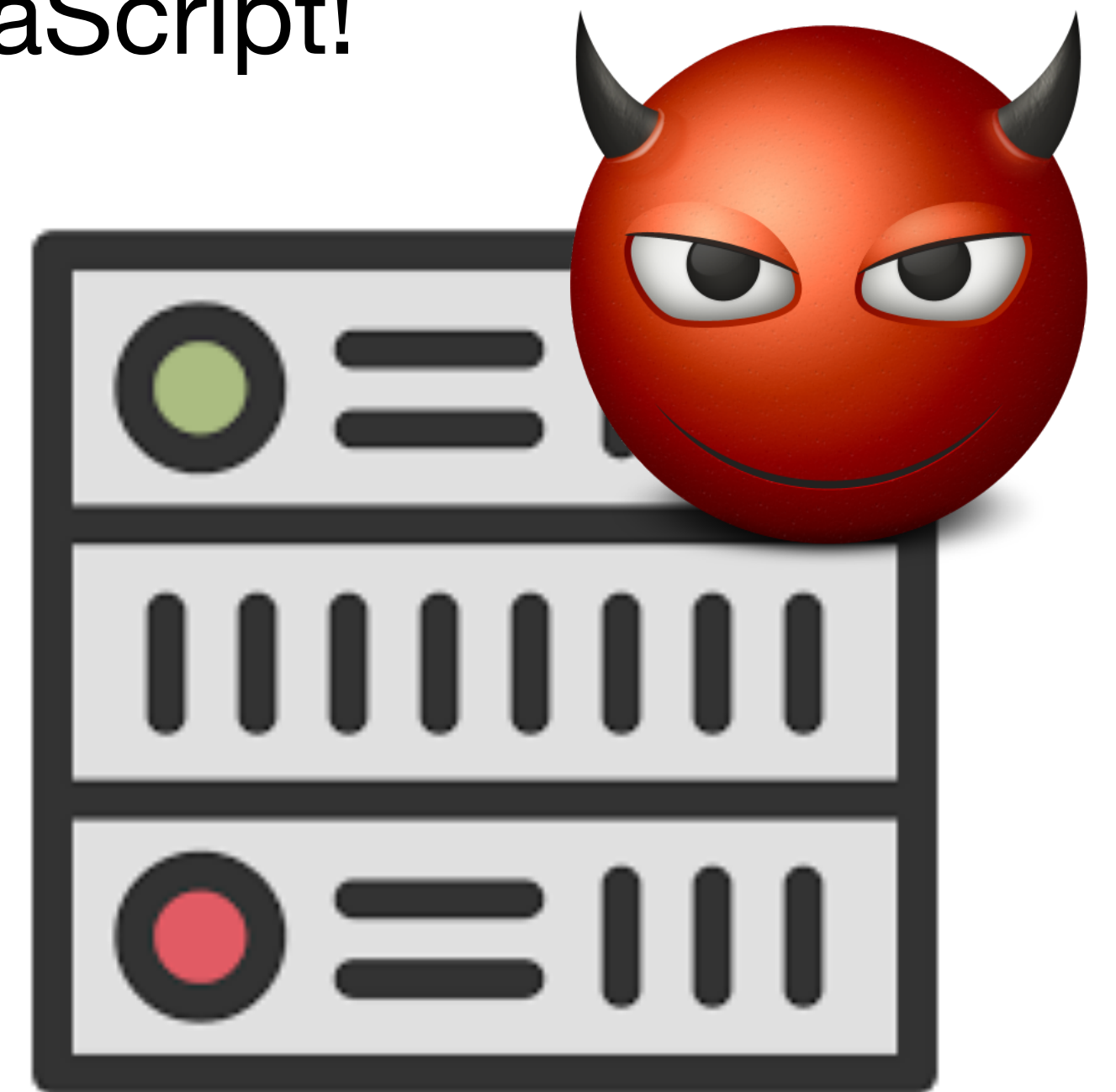


# Exemplo

- Servidor web malicioso pretende ganhar controlo da máquina de clientes:
  - E.g., para instalar malware, trojan, etc.
- Consegue executar código na máquina do cliente: JavaScript!



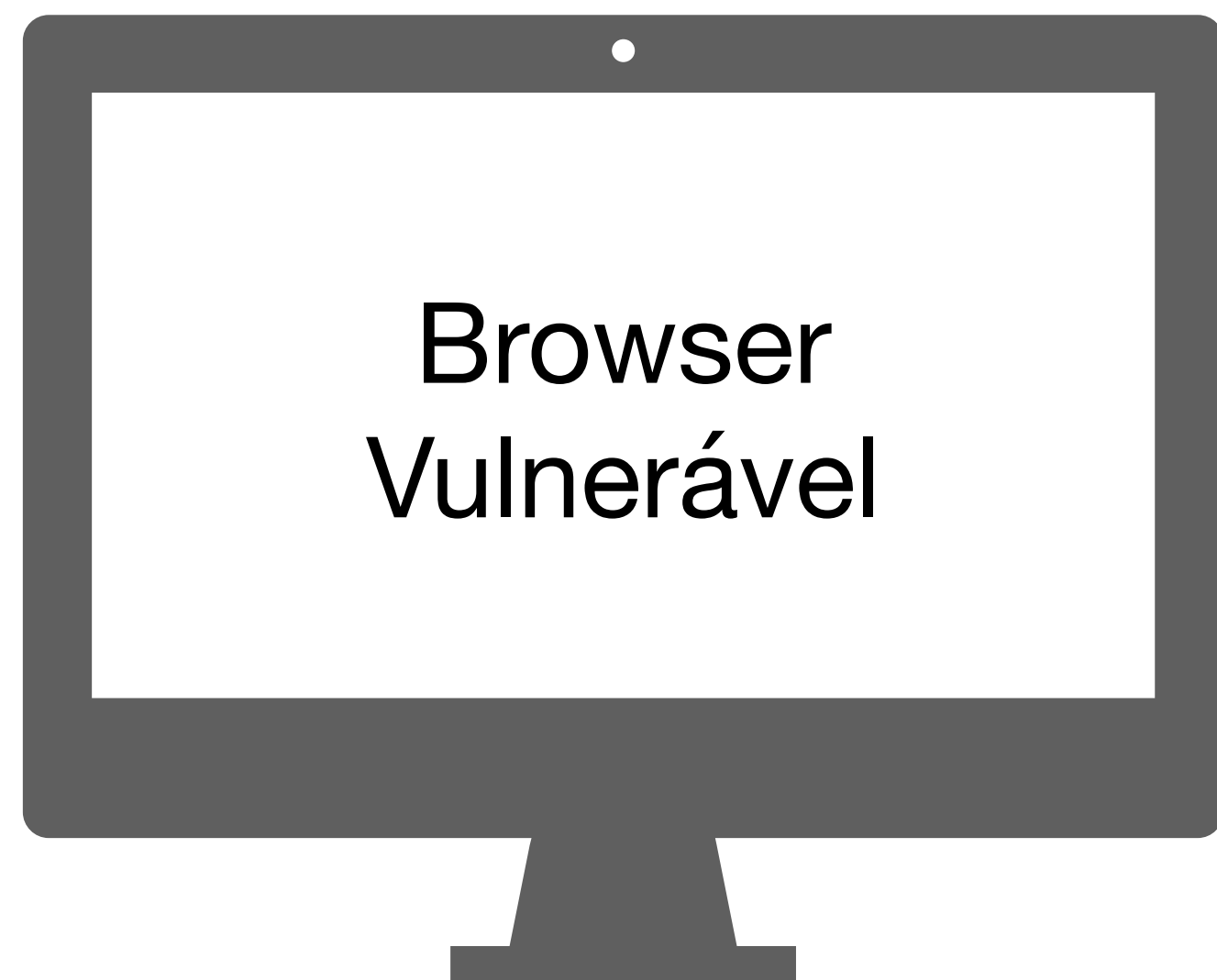
JavaScript malicioso



# Exemplo



- Problema:
  - o comportamento do browser depende de muitos fatores
  - o posicionamento na memória de código injetado é difícil/impossível de prever

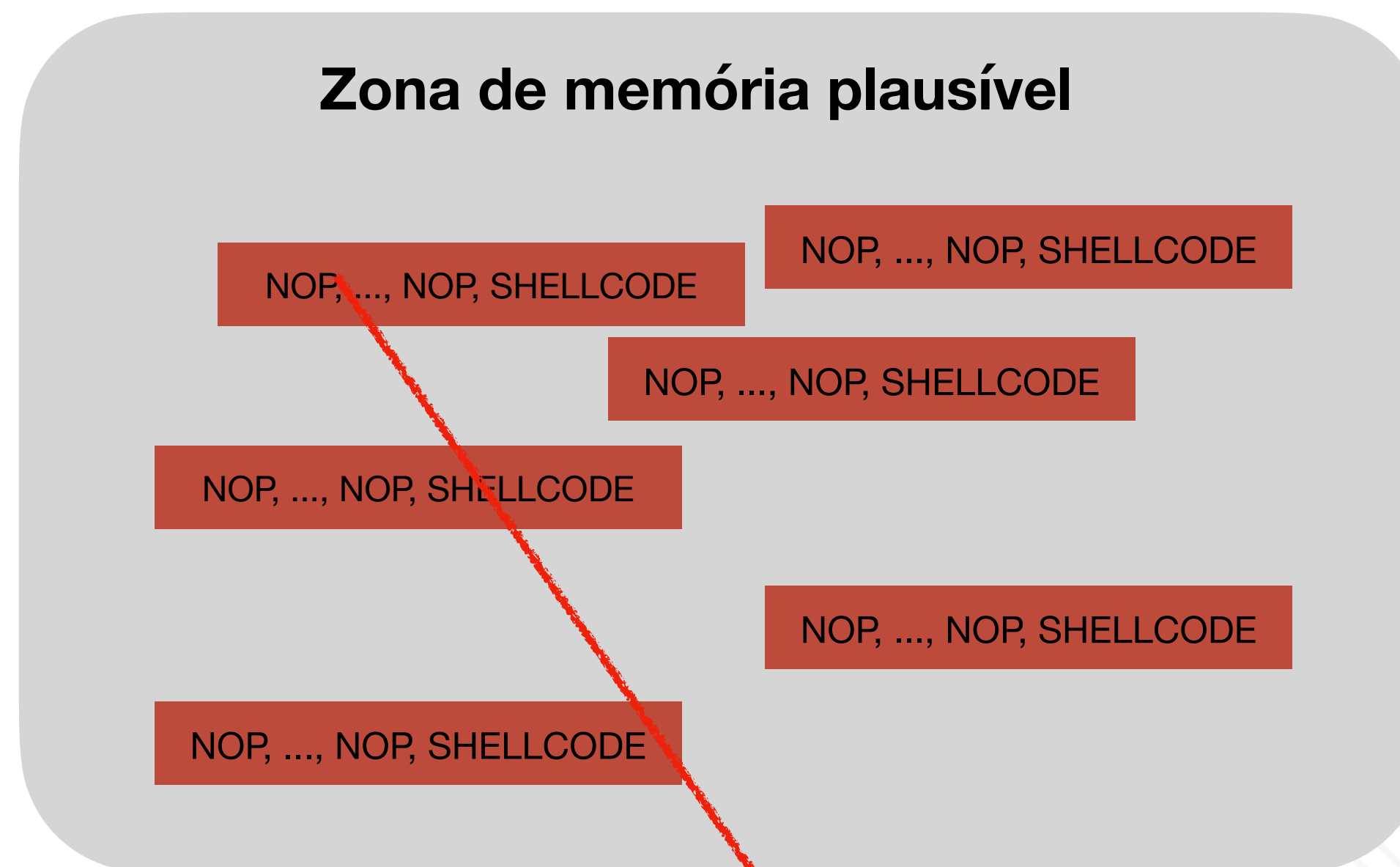


JavaScript malicioso




# Exemplo

- Os adversários fazem batota: "heap spraying"
- O JavaScript malicioso inunda a memória com cópias de shellcode
- O buffer overflow coloca o apontador para uma função alíngues na zona alvo
- Aumenta probabilidade de obter controlo



# Exemplo

## Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability

Title : Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability  
Advisory ID : FrSIRT/ADV-2005-0935  
CVE ID : CVE-2005-2087  
Rated as : **Critical**   
Remotely Exploitable : Yes  
Locally Exploitable : Yes  
Release Date : 2005-07-01

### Advisory Details

- ▶ [Description](#)
- ▶ [Affected Products](#)
- ▶ [Solution](#)
- ▶ [References](#)

### Technical Description



A vulnerability was identified in Microsoft Internet Explorer, which could be exploited by remote attackers to execute arbitrary commands. This flaw is due to an error in the "javaprxy.dll" COM Object when instantiated in Internet Explorer via a specially crafted HTML tag, which could be exploited via a malicious Web page to compromise and take complete control of a vulnerable system.

Proof of concept Exploit :

<http://www.frsirt.com/exploits/20050702.iejavaprxyexploit.pl.php>

### Credits

Vulnerability reported by Bernhard Müller and Martin Eiszner

### ChangeLog

2005-07-01 : Initial release  
2005-07-02 : Exploit available  
2005-07-05 : Updated CVE  
2005-07-05 : Updated Solution (KB903235 tool)  
2005-07-12 : Updated Solution (MS05-037)



# Use After Free

- Em linguagens orientadas a objetos (C++)
  - uma instância guarda endereços de métodos
- E se o código chama um método de uma instância destruída?
  - Talvez não haja crash se a memória permanecer estável
- Mas se entretanto o adversário conseguir preencher essa zona de memória com endereços à sua escolha?
  - Alocação de um input do adversário => execução de shellcode



# Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.



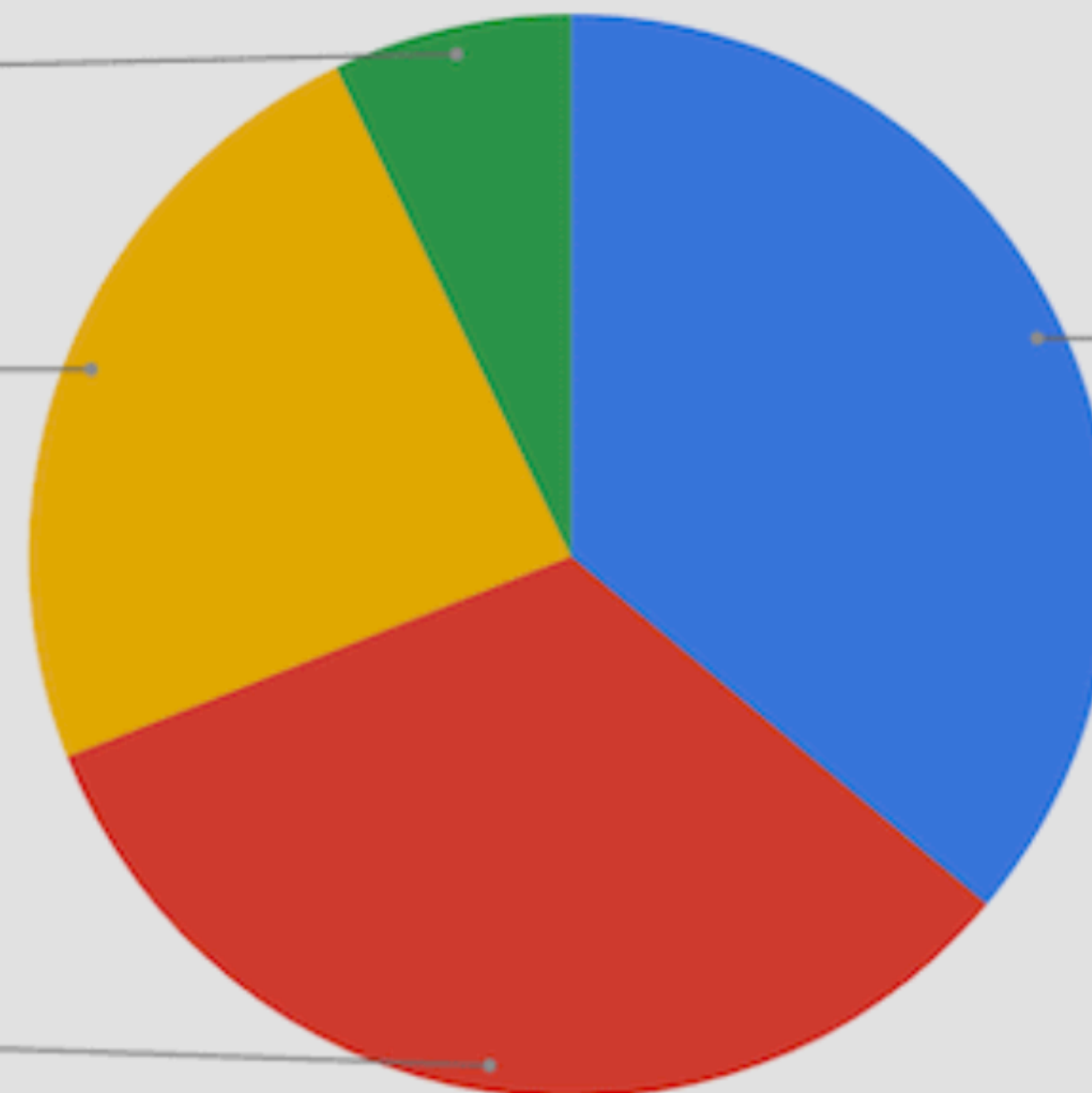
By [Catalin Cimpanu](#) for [Zero Day](#) | May 23, 2020 -- 06:00 GMT (07:00 BST) | Topic: [Security](#)

High+, impacting stable

Security-related assert  
7.1%

Other  
23.9%

Other memory unsafety  
32.9%



Use-after-free  
36.1%

# Outros tipos de overflow

# Overflow de Inteiros

- Quando a representação de inteiros do processador causa perda de informação
- Caso mais simples: truncatura por passagem para tipo mais pequeno

```
int i = 0x12345678;  
short s = i;  
char c = i;
```

# Overflow de Inteiros

- Casos mais subtis: aqui a forma de representar tamanhos é diferente.

```
struct s {  
    unsigned short len;  
    char buf[];  
};
```

```
void foo(struct s *p) {  
    char buffer[100];  
    if (p->len < sizeof buffer)  
        strcpy(buffer, p->buf);  
    // Use buffer  
}
```

```
int main(int argc, char *argv[]) {  
    size_t len = strlen(argv[0]);  
    struct s *p = malloc(len + 3);  
    p->len = len;  
    strcpy(p->buf, argv[0]);  
    foo(p);  
    return 0;  
}
```

pode não caber >>>

pode passar mesmo com strlen(p->buf) >> 100

# Overflow de Inteiros

- A aritmética pode dar resultados inesperados por overflow.

```
unsigned int product = a * b;  
unsigned int sum = a + b;  
unsigned int difference = a - b;
```

- Por exemplo, o que acontece em `0xff + 0x1` sobre `char`?
- Se estivermos a calcular o tamanho da região de memória podemos alocar 0 bytes em vez de 256 bytes!



# Overflow de Inteiros

- Noutros casos a comparação esquece a representação com sinal

```
if (x < 100) // segue código que usa x
```

- E se x for 0x f f f f f f f f? Este valor representa -1 em palavras de 32 bits!
- Mais uma vez podemos alocar uma região de memória demasiado pequena (ou vazia)
- Também pode haver problemas quando comparamos valores sem sinal e com sinal:

```
if (size < sizeof(x))  
    p = malloc(size); // size < 0
```

# O que faz o adversário

- Se controlar um inteiro no input pode provocar um overflow de inteiros num cálculo intermédio:
  - provoca a criação de um buffer demasiado pequeno => buffer overflow
  - utiliza as técnicas descritas anteriormente
  - ou, causar um crash é um ataque valido! (DoS)
- Caso típico:
  - tamanho a alocar: `expressao(y)`
  - `y` é controlado pelo atacante
  - `expressao(y) <= 0` por overflow

# Exemplo no openSSH

```
nresp = packet_get_int();  
if (nresp > 0) {  
    response = xmalloc(nresp*sizeof(char*));  
    for (i = 0; i < nresp; i++)  
        response[i] = packet_get_string(NULL);  
}
```

- Atacante controla `nresp`
- Overflow na multiplicação leva a alocação de 0 bytes

# ImageMagick Integer/Buffer Overflows in Processing XCF and Sun Bitmap Images Lets Remote Users Execute Arbitrary Code

**SecurityTracker Alert ID:** 1016749

**SecurityTracker URL:** <http://securitytracker.com/id/1016749>

**CVE Reference:** [CVE-2006-3743](#), [CVE-2006-3744](#) *(Links to External Site)*

**Date:** Aug 24 2006

**Impact:** [Execution of arbitrary code via network](#), [User access via network](#)

**Fix Available:** Yes **Vendor Confirmed:** Yes

**Version(s):** 6.2.9 and prior versions

**Description:** A vulnerability was reported in ImageMagick. A remote user can cause arbitrary code to be executed on the target user's system.

A remote user can create a specially crafted XCF or Sun bitmap graphic file that, when loaded by the target user, will trigger a buffer overflow and execute arbitrary code on the target system.

**Impact:** A remote user can create an image file that, when loaded by the target user, will execute arbitrary code on the target user's system.

**Solution:** The vendor has issued a fixed version (6.2.9-1), available at:

# Strings de formatação

- Todos conhecemos a família `printf`
- Recebe como primeiro parametro uma string de formatação:

```
printf("Isto é um inteiro %d e isto uma string %s\n", i, s)
```

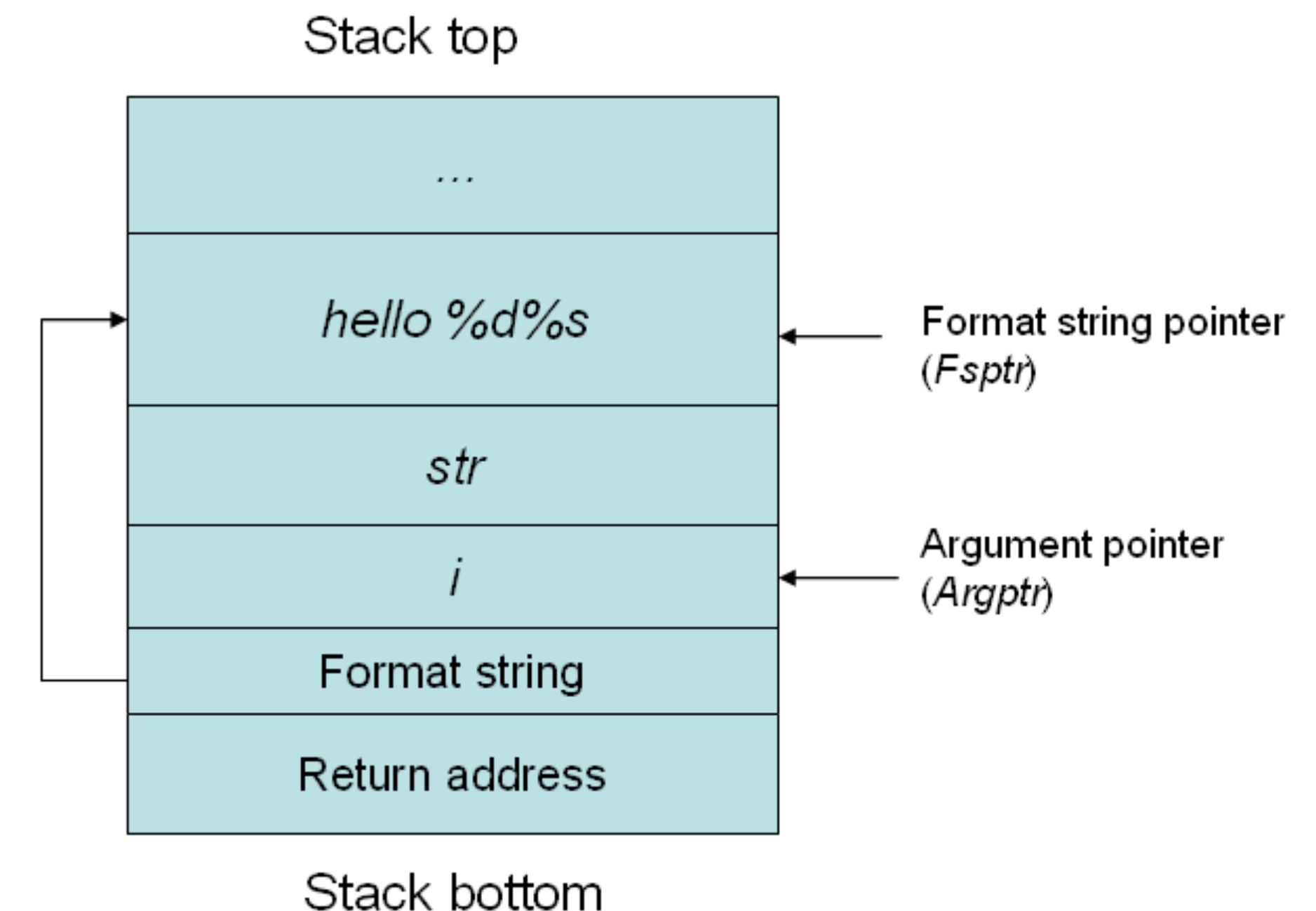
- E depois quantos parâmetros mais?
  - Depende ...
  - Mas como se implementa isto?



# Strings de formatação

```
printf("Isto é um inteiro %d e isto uma string %s\n", i, s)
```

- Os parâmetros são colocados na stack:
  - string de formatação: array de caracteres na própria stack (variável local)
  - inteiro *i* , string *s* (endereço)
  - endereço da string de formatação
- `printf` percorre string de formatação **e vai lendo valores cada vez mais acima na stack!**
- O que acontece se houver argumentos a menos/a mais?



# Strings de formatação

- Suponhamos um possível erro:

```
printf("%s\n", s)
```

```
printf(s) // esqueceu-se a string de formatação
```

O que vai acontecer?

- Se o atacante controlar a string de formatação:
  - `printf("%d\n")` imprime algo diretamente da stack
  - `printf("%s\n")` imprime algo apontado por endereço na stack
  - `printf("<endereco>%d%d%d...%s")` permite ler de qualquer <endereco> na memória! ==> **A própria string de formatação está na stack!!!**

# Strings de formatação

- Funciona apenas para leitura?
- Não: `if (strlen(src) < sizeof(dst)) sprintf(dst, src);`
- `sprintf` escreve para um array em memória
- E se `src` for controlado por adversário e contiver "%s"?
- Qual o tamanho da região escrita? ==> o tamanho da string a imprimir!

# Strings de formatação

n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.
---	---

- Existe uma opção de formatação que é radical: %n
- A ideia é guardar no endereço apontado pelo argumento seguinte o número de caracteres impressos até ao momento.
- Esta opção está excluída em grande parte das plataformas:
  - Se o adversário controlar a string de formatação consegue escrever para posições arbitrárias em memória.

# Como evitar estes ataques?

