

Fundamentos de Segurança Informática (FSI)

2022/2023 - LEIC

Manuel Barbosa
mbb@fc.up.pt

Hugo Pacheco
hpacheco@fc.up.pt

Aula 5

Segurança de Software:

Controlo (Parte 3)

Controlo está mais difícil

- O que é necessário:
 - **Injetar código malicioso** na memória (shellcode)
 - **Alterar controlo de execução** para saltar para essa zona: done!
- Sistemas modernos: **Data Execution Prevention (DEP)** ou **W ^ X (Write xor Execute)**
 - Uma página de memória executável não pode ser escrita
 - Uma página de memória que pode ser escrita não pode ser executável
 - Algumas exceções em casos de “Just-In-Time” Compilation (👹)
- Sistemas modernos: **Address Space Layout Randomization (ASLR)**
 - A localização em memória de partes críticas da memória de um programa é "baralhada" em cada execução

JIT Spraying

- Compilação “Just-In-Time”: comum para bytecode, e.g., Java, JavaScript, Adobe Flash
- Código mais utilizado é compilado para código máquina “on-the-fly”
- Abusar do compilador JIT para inserir código malicioso em runtime:
 - compilar constantes + **saltar para “meios endereços”!**
 - código executável ⇒ **DEP** ☠
 - spray de shellcode ⇒ **ASLR** ☠

Instruction code injection via **ActionScript**

```
var ret=(0x3C909090^0x3C909090^0x3C909090^0x3C909090);
```

↓ ↓ ↓

0x1A1A0100:	B89090903C	MOV EAX,	3C909090	} Executable
0x1A1A0105:	359090903C	XOR EAX,	3C909090	
0x1A1A010A:	359090903C	XOR EAX,	3C909090	
0x1A1A010F:	359090903C	XOR EAX,	3C909090	

↓ ↓ ↓ + 0x01 to address

0x1A1A0101:	90	NOP	} As I said – executable
0x1A1A0102:	90	NOP	
0x1A1A0103:	90	NOP	
0x1A1A0104:	3C35	CMP AL, 35	
0x1A1A0106:	90	NOP	
0x1A1A0107:	90	NOP	
0x1A1A0108:	90	NOP	
0x1A1A0109:	3C35	CMP AL, 35	

Reutilização de Código

- Pensar como um atacante:
 - se não podemos injetar o nosso código
 - temos de reutilizar código que já está em memória executável
- **bibliotecas!**



Bibliotecas!

- E.g. Google Chrome:
 - utiliza perto de 100 bibliotecas
- A libc é utilizada por (quase) todos os programas
- Contém funções úteis:
 - `system`: correr comando shell
 - `mprotect`: alterar permissões de memória
- Cujos endereços são mais fáceis de prever



Usar libc como shellcode

- Utilizam-se as técnicas apresentadas nas aulas anteriores
- Mas o endereço de retorno será o de uma função da biblioteca
- Eg, se quisermos usar a função `system` ou `mprotect` temos de configurar a stack de acordo com o que essas funções necessitam:

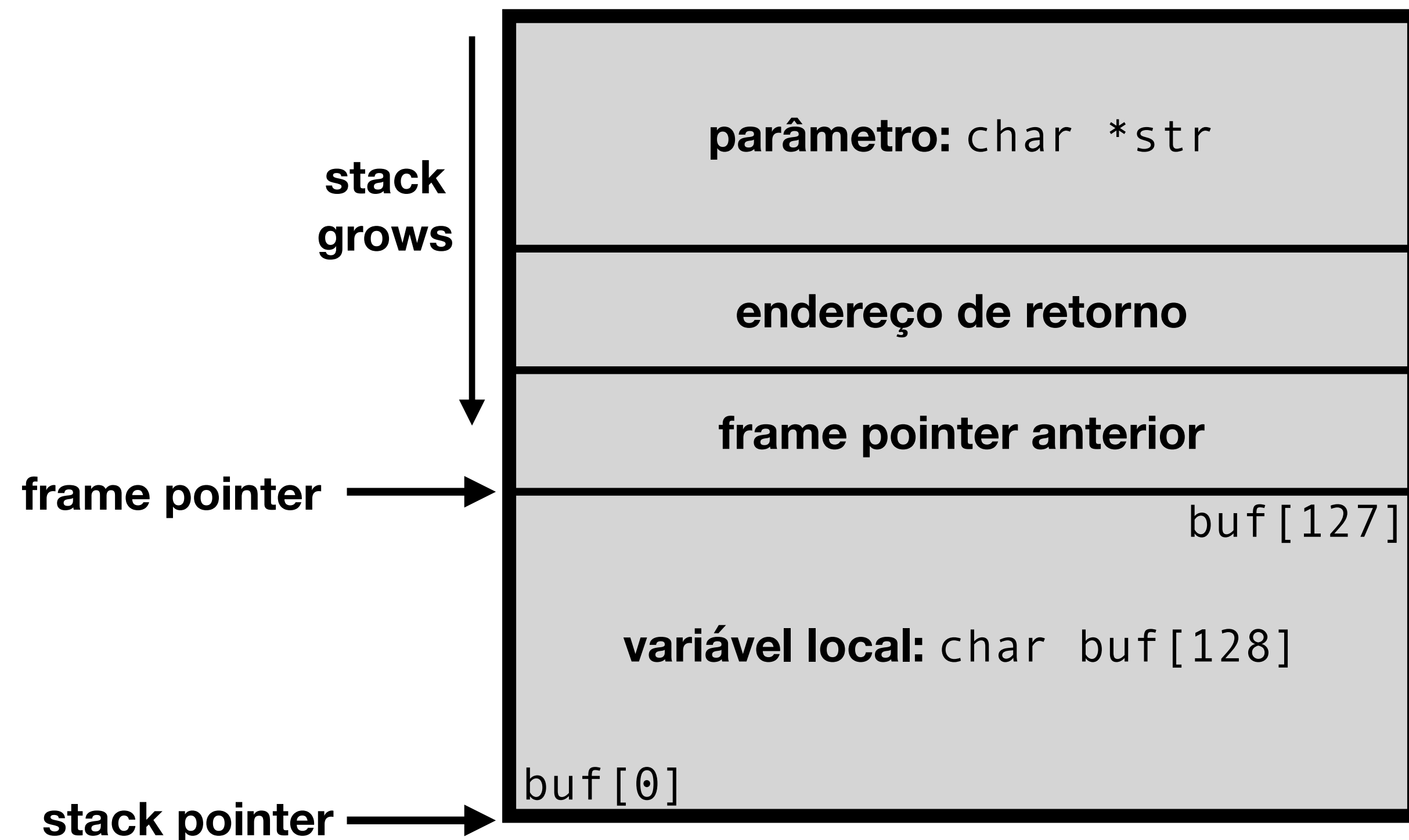
```
int system(const char *command);  
int mprotect(void *addr, size_t len, int prot);
```

- É preciso perceber em detalhe como funciona a utilização da stack
- Os exemplos que vamos ver são versões simplificadas

Flashback: Stack Smashing

- Quando entramos nesta função, a stack frame tem o seguinte aspecto:

```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    usar(buf);  
}
```

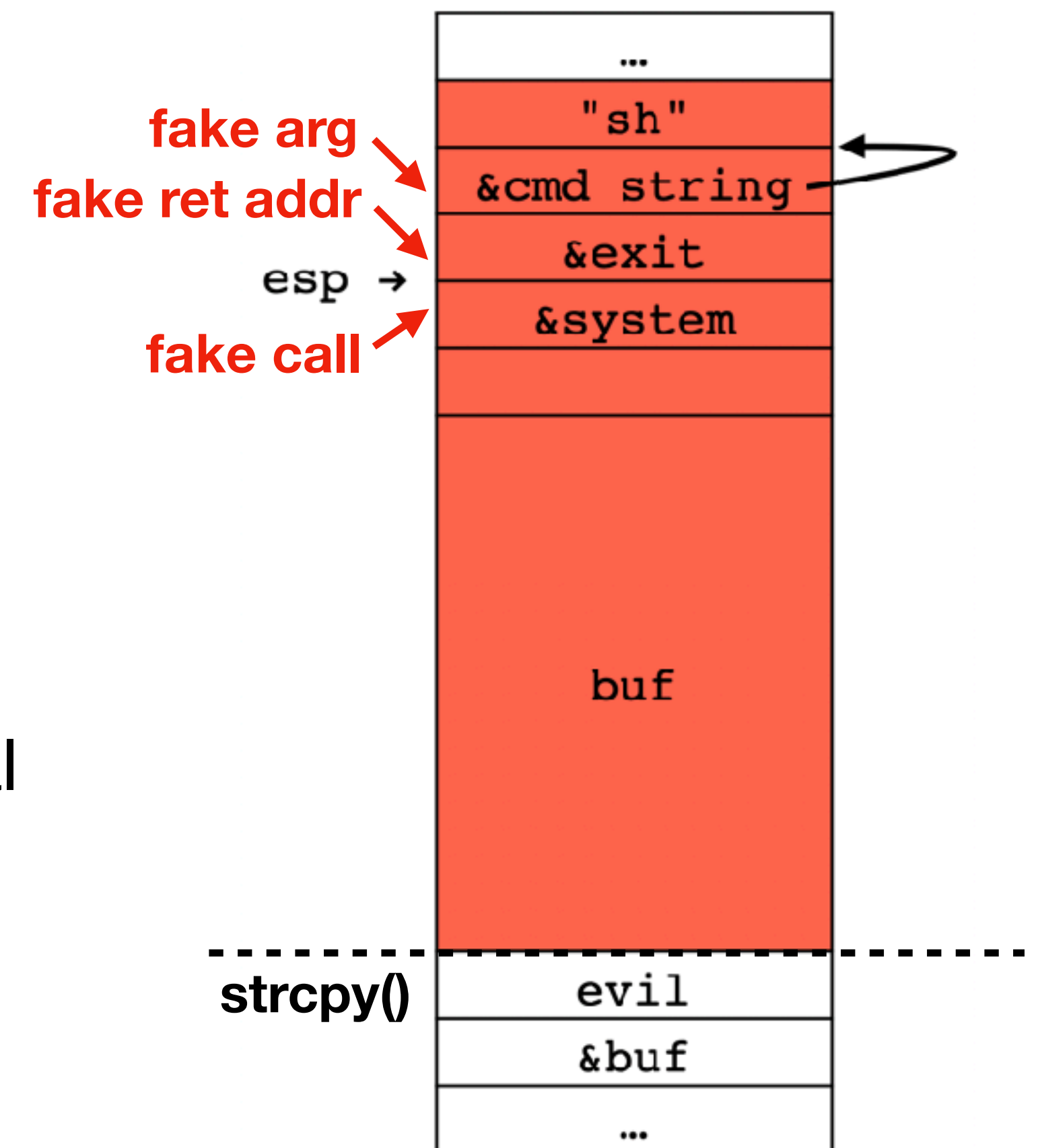


- A função que faz a chamada armazena parâmetros e endereço de retorno
- Quando se entra na função é armazenado o frame pointer anterior e criado espaço para variáveis locais
- Cada função limpa o espaço que cria na stack:** o processador depois utiliza o endereço de retorno automaticamente, e **a função que fez a chamada limpa os parâmetros**

Como chamar `system`?

- Os dados injetados têm de:
 - substituir o endereço de retorno da função actual por `&system`
 - configurar a stack para uma entrada correta na função `system`:
 - o endereço de uma string com o comando shell (parâmetro)
 - um endereço de retorno para quando `system` terminar
- Para evitar um crash (porquê?): **⇒ evitar deteção**
 - Colocar string de shell para cima na stack longe da frame actual
 - Retornar para função que encerra programa sem erro e.g., `exit(0)`

```
void foo(char *evil) {  
    char buf[32];  
    strcpy(buf, evil);  
}
```



Como chamar mprotect?

- Os dados injetados têm de:

```
int mprotect(void *addr,  
             size_t len,  
             int prot);
```

- substituir o endereço de retorno da função actual por &mprotect

- configurar a stack para uma entrada correta na função mprotect:

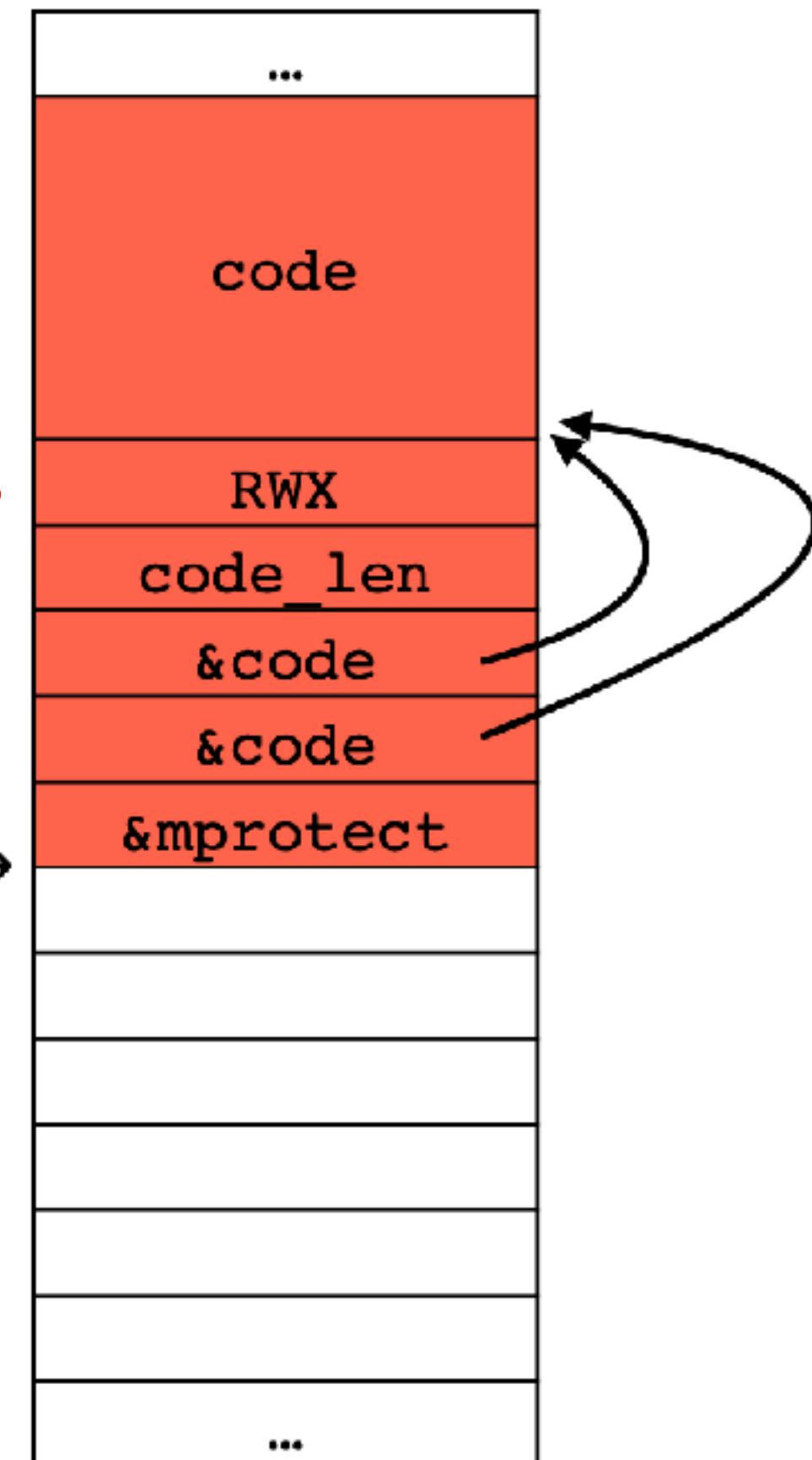
- o endereço de memória onde está o shellcode injetado, comprimento e permissões (parâmetros)

- um endereço de retorno para quando mprotect terminar ⇒ shellcode

- Não é fácil porque os parâmetros a passar a mprotect (endereço) geralmente contêm '\0', o que reduz as possibilidades de exploit (não funciona com strcpy, mas talvez com memcpy)

new permissions
memblock size
memblock addr
ret addr

esp →



**Esta técnica pode ser generalizada:
return-oriented programming**

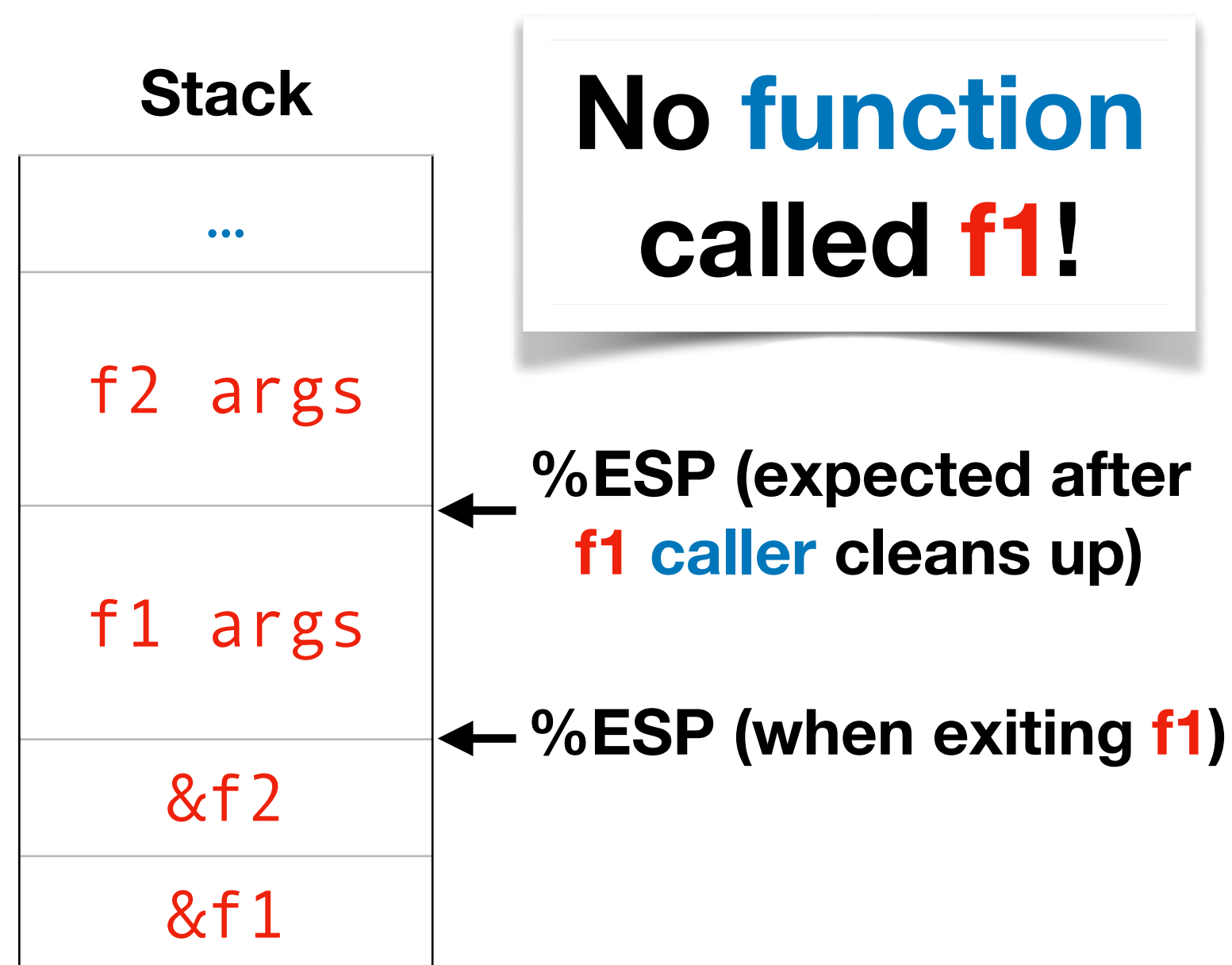
Reutilização de Código

- Pensar como um atacante:
 - se conseguimos executar uma função de uma biblioteca ...
 - será que conseguimos chamar N funções em sequência?



Mais complexo, mas possível

- É necessário construir toda uma estrutura na stack que esteja preparada para a sequência de funções:
 - retornamos para a primeira função a chamar
 - parâmetros + endereço de retorno = segunda função a chamar



- colocámos na stack os parâmetros para a primeira função, mas ...
- A primeira função não limpa!
- A segunda função não é responsável pela limpeza da stack
- quando entramos nessa função a stack contém ainda "lixo" da chamada anterior

Casos mais simples

- Se uma função não recebe parâmetros, a função que a chama apenas armazena na stack o endereço de retorno!
- Se quisermos que N funções sem parâmetros sejam executadas em sequência, basta colocar os seus endereços por ordem na stack
- A função f1 retorna para f2, que retorna para f3, etc

Stack

...
&f3
&f2
&f1

Casos mais simples

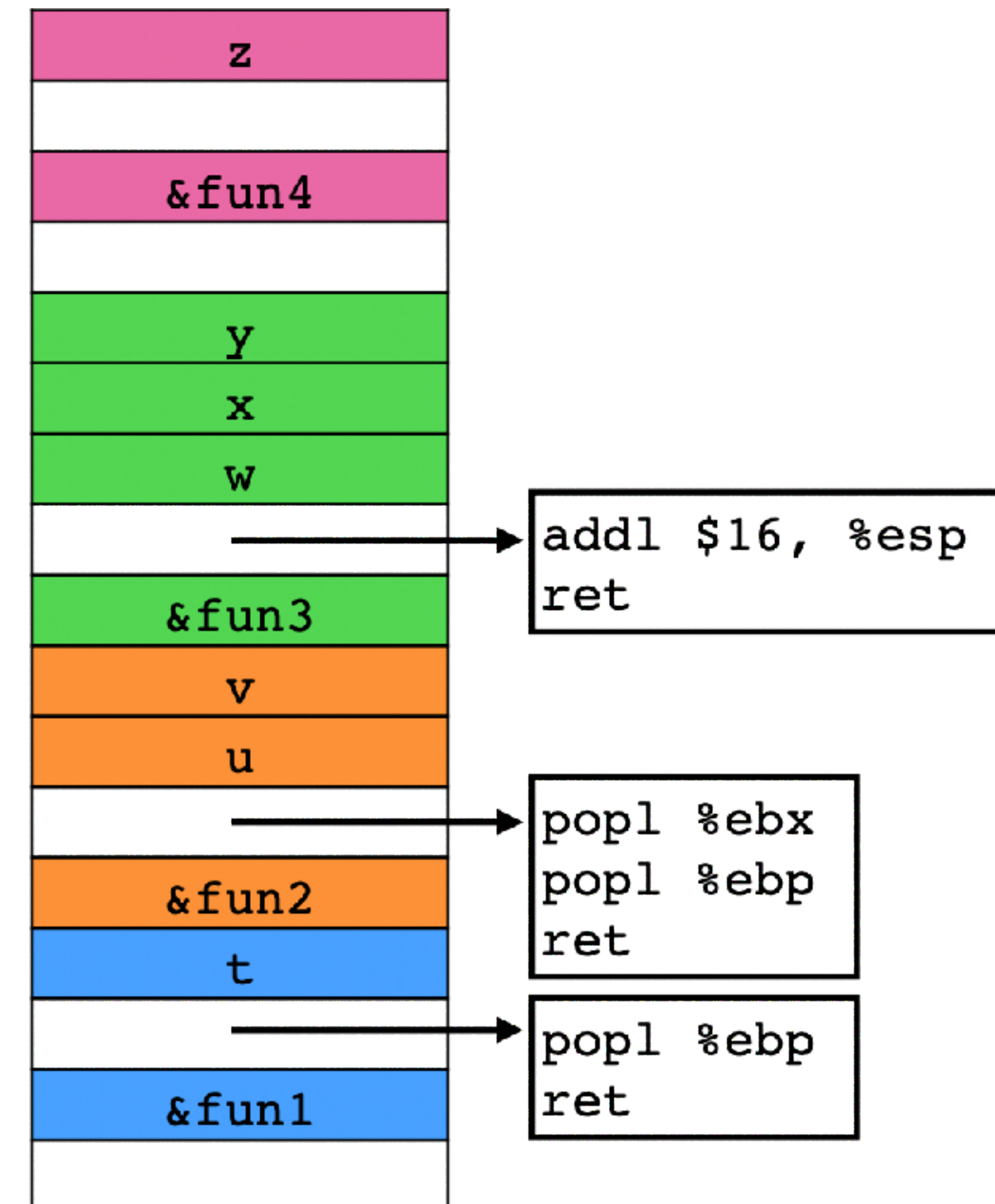
- Depois de uma sequência de qualquer tamanho de funções que não recebem parâmetros
- Podemos chamar uma função que recebe parâmetros ...
- e terminar chamando uma função que não recebe parâmetros

Stack

...
f_n param 3
f_n param 2
f_n param 1
$\&f_{n+1}$
...
$\&f_3$
$\&f_2$
$\&f_1$

Casos mais complexos

- Existem outras combinações que funcionam neste estilo
- Mas e se quisermos chamar uma sequência arbitrária?
 - `f1(t); f2(u,v); f3(w,x,y); ...`
- É necessário "criar" versões alternativas destas funções que limpam os seus próprios parâmetros da stack quando retornam. Como?
 - colocamos um outro endereço na stack entre as duas funções `f1` e `f2`
 - esse endereço é de um pedaço de código arbitrário que:
 - retira da stack o número de bytes necessário (por `pop` ou alteração direta do stack pointer)
 - retorna para a função seguinte => `f2`!
- Esta técnica pode utilizar-se para encadear um número arbitrário de chamadas



Porquê parar aqui?

- As pequenas sequências que estamos a utilizar para limpar a stack serão especiais?
 - `pop <reg>; ret`
- Em geral, se eu encontrar algures no código em memória:
 - `address1: i1; i2; i3; ret`
 - `address2: i4; i5; i6; ret`
- E colocar na stack `address1` e `address2` e retornar para o primeiro:
 - serão executadas `i1; i2; i3; i4; i5; i6; ret`

Return Oriented Programming

- Considerar o assembly de código em memória (programa ou biblioteca) como uma "sopa de letras"
- Identificar sequências de código úteis (geralmente terminando em `ret`, mais recentemente também sequências terminando em `jmp` ou `call`)
- Coletar essas sequências como "gadgets" reutilizáveis
- "Colar" os gadgets como uma sopa de letras



Return-Oriented Programming

is A lot like a ransom
note, BUT instead of cutting
cut Letters from Magazines,
YOU ARE cutting out
instructions from text
segments

Image by Dino Dai Zovi

Que operações se podem fazer?

- Depende do código a que temos acesso
- Operações aritméticas/lógicas e tudo o que seja "straight-line" é mais simples (nomeadamente shellcode):
 - composição sequencial de pequenos blocos de código
 - pesquisar de forma automática no código alvo um conjunto de gadgets
 - quando compostos devem ser **equivalentes** ao código que pretendemos
- Conceptualmente Turing-complete!

Um exemplo mais complexo

- Como fazemos um salto condicional?
 - temos uma condição c que queremos avaliar
 - queremos retornar para uma função f se c for verdadeira
 - queremos retornar para uma função g se for falsa
- A solução tem de passar por alterar o stack pointer tal que:
 - se c é verdadeira, aponta para uma sequência que começa em $\&f$
 - se c é falsa, aponta para uma sequência que começa em $\&g$

Um exemplo mais complexo

- A solução tem de passar por alterar o stack pointer tal que:
 - se c é verdadeira, aponta para uma sequência que começa em $\&f$
 - se c é falsa, aponta para uma sequência que começa em $\&g$
- Podemos implementar esta operação da seguinte forma:
 - utilizamos uma sequência de gadgets para inicializar um registo m a $0x00000000$ se c é verdadeiro e $0xffffffff$ se c é falsa
 - colocamos o stack pointer a apontar para a sequência que começa em $\&f$
 - calculamos um offset o que temos que adicionar ao stack pointer para que fique a apontar para a sequência que começa em $\&g$
 - calculamos $sp = m \&\& o + sp$ usando um conjunto de gadgets adequado

Timeline em Retrospectiva

- Nos primeiros tempos sem DEP ou ASLR => injeção de código
- Com DEP => ROP aparece em 2007 com gadgets da libc
- Rapidamente surgem ferramentas para automatizar ROP
- Com ALSR => ROP fixando-se em código que permanece em endereços previsíveis
- Hoje em dia: todos os SO/compiladores usam position-independent executables (PIO) que tornam ROP muito mais difícil
- Técnicas avançadas de identificação de gadgets em runtime!

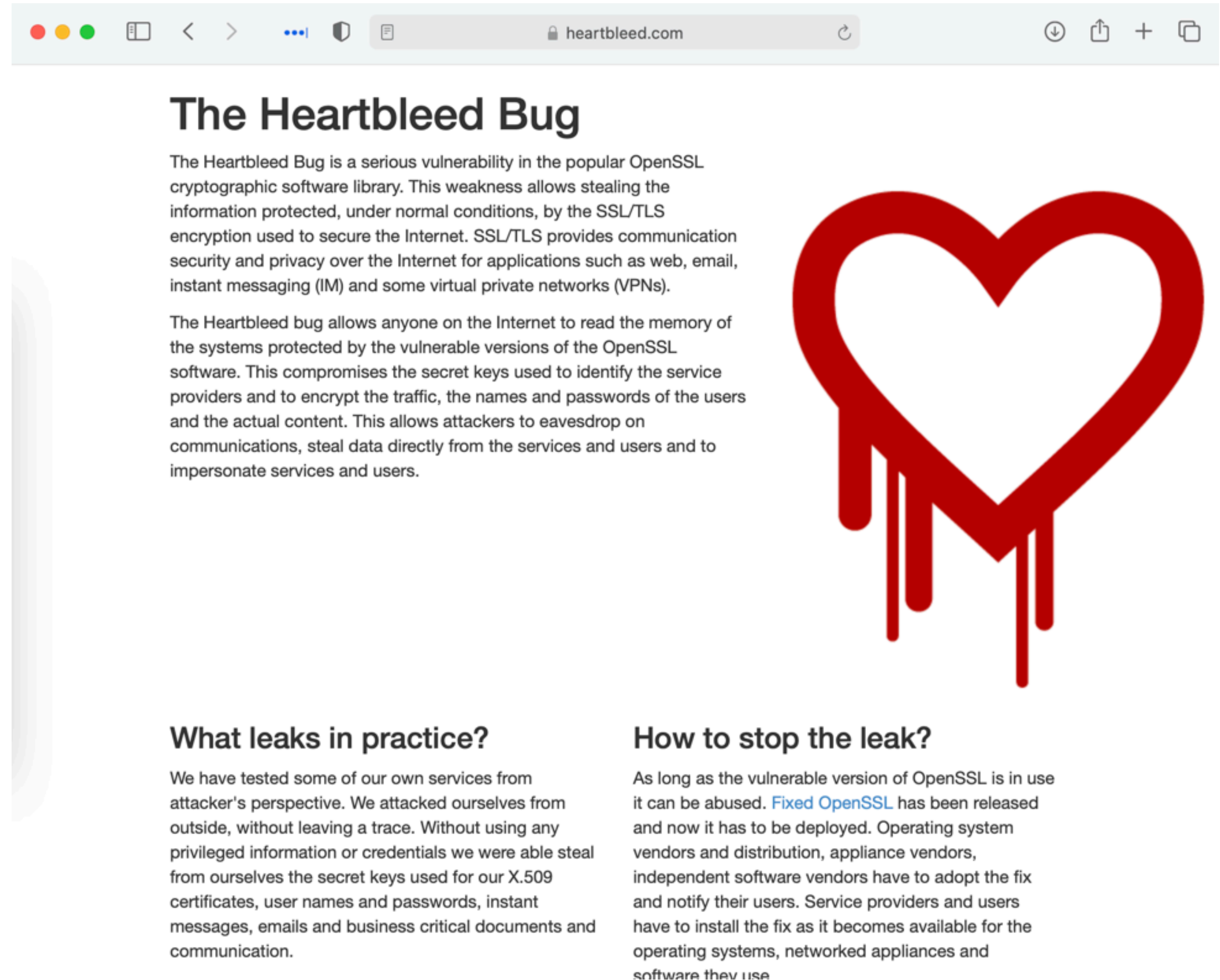
Wrap-up de Buffer Overflows

O que vimos

- Se for possível a um atacante **influenciar um input**:
 - é essencial garantir que **todos os acessos a memória são corretos**
- Se um atacante conseguir **escrever fora da zona de memória** reservada para essa finalidade:
 - pode alterar o controlo de fluxo do programa
 - utilizar o programa contra a própria plataforma onde está a correr
- E se o atacante conseguir só **ler fora da zona correta**?

HeartBleed

- "heartbeat" TLS: cliente envia pequena mensagem ao servidor para manter a sessão viva
- mensagem contém um inteiro (fixo no standard) que descreve o comprimento dos dados a transmitir
- implementação reutiliza o valor fornecido pelo cliente para contar quantos bytes tem de ler de memória
- resultado: cliente malicioso obtém uma resposta com informação sensível que está na memória do servidor (e.g., passwords, chaves criptográficas, etc.)



The screenshot shows a web browser window with the address bar displaying "heartbleed.com". The page title is "The Heartbleed Bug". The main text describes the vulnerability in OpenSSL, stating that it allows stealing of information protected by SSL/TLS encryption. It explains that the bug allows anyone on the Internet to read the memory of systems protected by vulnerable versions of OpenSSL, compromising secret keys, user names, passwords, and other sensitive data. To the right of the text is a large red heart icon with red liquid dripping down from its base. Below the main text, there are two sections: "What leaks in practice?" and "How to stop the leak?". The "What leaks in practice?" section describes how the bug was tested on the company's own services, showing that sensitive information like X.509 certificates, user names, passwords, and business documents could be stolen. The "How to stop the leak?" section states that as long as the vulnerable version of OpenSSL is in use, it can be abused, and that a fixed version has been released and must be deployed by operating system vendors, appliance vendors, and independent software vendors.

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.


What leaks in practice?

We have tested some of our own services from attacker's perspective. We attacked ourselves from outside, without leaving a trace. Without using any privileged information or credentials we were able to steal from ourselves the secret keys used for our X.509 certificates, user names and passwords, instant messages, emails and business critical documents and communication.

How to stop the leak?

As long as the vulnerable version of OpenSSL is in use it can be abused. [Fixed OpenSSL](#) has been released and now it has to be deployed. Operating system vendors and distribution, appliance vendors, independent software vendors have to adopt the fix and notify their users. Service providers and users have to install the fix as it becomes available for the operating systems, networked appliances and software they use.

Dezenas de milhar de vulnerabilidades



[CVE List](#)[CNAs](#)[WGs](#)[Board](#)[About](#)

[Search CVE List](#)[Downloads](#)[Data Feeds](#)[Update a CVE Record](#)[Request](#)

TOTAL CVE Records: **156238**

HOME > CVE > SEARCH RESULTS

Search Results

There are **11753** CVE Records that match your search.

Name	Description
CVE-2021-36089	Grok 7.6.6 through 9.2.0 has a heap-based buffer overflow in grk::FileFormatDecompress::apply_palette_clr (called from
CVE-2021-36083	KDE KImageFormats 5.70.0 through 5.81.0 has a stack-based buffer overflow in XCFImageFormat::loadTileRLE.
CVE-2021-36082	ntop nDPI 3.4 has a stack-based buffer overflow in processClientServerHello.
CVE-2021-35474	Stack-based Buffer Overflow vulnerability in cachekey plugin of Apache Traffic Server. This issue affects Apache Traffic Ser
CVE-2021-3507	A heap buffer overflow was found in the floppy disk emulator of QEMU up to 6.0.0 (including). It could occur in fdctrl_trar data transfers from the floppy drive to the guest system. A privileged guest user could use this flaw to crash the QEMU pr information leakage from the host memory.
CVE-2021-3496	A heap-based buffer overflow was found in jhead in version 3.06 in Get16u() in exif.c when processing a crafted file.
CVE-2021-3482	A flaw was found in Exiv2 in versions before and including 0.27.4-RC1. Improper input validation of the rawData.size prop to a heap-based buffer overflow via a crafted JPG image containing malicious EXIF data.
CVE-2021-34813	Matrix libolm before 3.2.3 allows a malicious Matrix homeserver to crash a client (while it is attempting to retrieve an Olm olm_pk_decrypt has a stack-based buffer overflow. Remote code execution might be possible for some nonstandard build
CVE-2021-3466	A flaw was found in libmicrohttpd in versions before 0.9.71. A missing bounds check in the post_process_urlencoded func write arbitrary data in an application that uses libmicrohttpd. The highest threat from this vulnerability is to data confiden
CVE-2021-34557	XScreenSaver 5.45 can be bypassed if the machine has more than ten disconnectable video outputs. A buffer overflow in standard screen lock authentication mechanism by crashing XScreenSaver. The attacker must physically disconnect many
CVE-2021-34382	Trusty TLK contains a vulnerability in the NVIDIA TLK kernel's tz_map_shared_mem function where an integer ov the logging buffer to overflow, allowing writes to arbitrary addresses within the kernel.
CVE-2021-3438	A potential buffer overflow in the software drivers for certain HP LaserJet products and Samsung product printers could le
CVE-2021-34375	Trusty contains a vulnerability in all trusted applications (TAs) where the stack cookie was not randomized, which might re service, escalation of privileges, and information disclosure.
CVE-2021-34372	Trusty (the trusted OS produced by NVIDIA for Jetson devices) driver contains a vulnerability in the NVIDIA OTE protocol

Como são descobertas?

- Geralmente um erro na gestão de memória
- O primeiro sinal é um crash do programa
- Para procurar por vulnerabilidades:
 - utilizar um "fuzzer" para fornecer inputs cegamente a um programa
 - se o programa crashar, investigar porquê
- Se o overflow se verificar, é geralmente reconhecido imediatamente como uma potencial vulnerabilidade
- A construção de um exploit geralmente exige muito mais trabalho (como vimos)

Um crash pode ser valioso!

Jailbreakers use Apple crash reports to 'free' iPhones

Bug hunt

Hackers like Mr Hill hunt for programming errors, or bugs, in Apple's software. Bugs may result in a program crashing or shutting down, and they are like gold dust to hackers because sometimes they can be exploited to create a jailbreak.

To help prevent this, Apple's phones record details of program crashes and send these reports back to the company. Apple's programmers can then analyse the crash reports and fix any underlying bugs that pose serious security risks or that could be exploited to create a jailbreak.

But crash reporting causes particular problems for Mr Hill and his team. That is because the hackers may have to crash a particular program thousands of times as they work out how to exploit a bug successfully, Mr Hill says, and this alerts Apple that the bug exists and that hackers may be investigating it.

In September Mr Hill was working on exploiting five separate bugs found in early versions of Apple's iOS 5 software to create a full or "untethered" jailbreak, but the most important ones had been patched by Apple when the final version of its software was released in October. Crash reporting was probably to blame, he believes.

Crash reports

The solution to this problem is to subvert Apple's crash reporting capability by turning it against the company, he says.

"Chronic Dev is ready to turn this little information battle into an all-out, no-holds-barred information WAR," Mr Hill wrote on the Chronic-Dev blog recently, using his nom de guerre Posixninja.

To do this he has written and distributed a program called CDevreporter that iPhone users can download to their PC or Mac. The program intercepts crash reports from their phones destined for Apple and sends them to the Chronic Dev team.

If crash reports are like gold dust then Mr Hill and his team are now sitting on a gold mine.

"In the first couple of days after we released CDevreporter we received about twelve million crash reports," he says.

"I can open up a crash report and pretty much tell if it will be useful or not for developing a jailbreak, but we have so many that I am working on an automated system to help me analyse them."

Ethical hacking

- É crucial haver quem procure estas vulnerabilidades para termos sistemas mais seguros
- É importante ter muito cuidado com a forma como se divulgam essas vulnerabilidades
- Os sistemas de bug bounty e vulnerability reporting são desenhados para:
 - evitar que "zero-day vulnerabilities" detectadas por hackers bem intencionados acabem por causar danos

Um exemplo: SQLSlammer

- Buffer overflow identificado (e corrigido) no SQL Server algures em 2002
- Um investigador demonstra publicamente (blog “writeup”) como se poderia explorar esse bug para propagar um worm
- Esse código é depois utilizado para criar o SQLSlammer.
- Um número enorme de sistemas não tinha sido "patched"

The spread of SQL Slammer

At its height, SQL Slammer, which was the most widespread worm since 2001's [Code Red worm](#), doubled in size every 8.5 seconds. South Korea, one of the most connected countries in the world at the time, had an outage of internet and cell phone coverage for 27 million people, while in the US, almost all of Bank of America's 13,000 ATMs were [temporarily knocked offline](#).

<https://www.welivesecurity.com/2016/09/30/flashback-friday-sql-slammer/>
https://en.wikipedia.org/wiki/SQL_Slammer

Um exemplo: Blaster

Blaster (also known as **Lovsan**, **Lovesan**, or **MSBlast**) was a [computer worm](#) that spread on computers running [operating systems Windows XP](#) and [Windows 2000](#) during August 2003.^[1]

The worm was first noticed and started spreading on August 11, 2003. The rate that it spread increased until the number of infections peaked on August 13, 2003. Once a network (such as a company or university) was infected, it spread more quickly within the network because firewalls typically did not prevent internal machines from using a certain port.^[2] Filtering by ISPs and widespread publicity about the worm curbed the spread of Blaster.

Creation and effects [\[edit \]](#)

According to court papers, the original Blaster was created after security researchers from the [Chinese group Xfocus reverse engineered the original Microsoft patch](#) that allowed for execution of the attack.^[4]

The worm spreads by exploiting a [buffer overflow](#) discovered by the Polish security research group Last Stage of Delirium^[5] in the [DCOM RPC](#) service on the affected operating systems, for which a patch had been released one month earlier in [MS03-026](#)^[6] and later in [MS03-039](#)^[7]. This allowed the worm to spread without users opening attachments simply by spamming itself to large numbers of random IP addresses. Four versions have been detected in the wild.^[6] These are the most well-known exploits of the original flaw in RPC, but there were in fact another 12 different vulnerabilities that did not see as much media attention.^[7]

Como evitar estes ataques?



Acknowledgements

- This lecture's slides have been inspired by the following lectures:
 - CSE127: Low-level Software Security III: Integer overflow, ROP and CFI
 - CS155: Control Hijacking: Defenses
 - CS343: Code reuse attacks + Return-oriented programming