



```
# ./BINARY -c EXPLOITATION -v P.1
```

cat AGENDA

- ❖ profile
- ❖ objetivo
- ❖ Tools
- ❖ Contexto
 - ❖ Python?
 - ❖ Entendendo a arquitetura
 - ❖ O que é *Buffer Overflow*
 - ❖ Mitigações e meios de proteção
- ❖ Criando um shellcode em Assembly
- ❖ Explorando o *vanilla buffer overflow*

echo \$PROFILE

- ❖ Name: Leonardo Toledo
- ❖ AKA: H41stur
- ❖ Role: Pentester and Security Researcher, Data Scientist
- ❖ Blog: <https://h41stur.github.io/>
- ❖ GitHub: <https://github.com/h41stur>
- ❖ Material da Palestra ficará disponível em:
<https://github.com/h41stur/talks>

more .objetivo

- ❖ Esta talk não tem como objetivo ensinar conceitos sobre programação, nem aprofundamento em alguma linguagem específica.
- ❖ O esperado é que durante o decorrer da agenda, os participantes entendam o processo de análise e exploração de binários no sistema operacional Linux, assim como o *bypass* de mitigações e proteções comuns, e tenham noção das ferramentas e fundamentos de todo o processo.

less TOOLS

- ❖ Linux (pode ser qualquer distro)
- ❖ Debugger GDB
- ❖ Plugin Peda
- ❖ Python3
- ❖ Python2
- ❖ Bastante matemática hex

Contexto

Por que utilizar Python para explorar baixo nível?

- ❖ Sintaxe simples;
- ❖ Rico em bibliotecas;
- ❖ Linguagem interpretada;
- ❖ Orientação a objetos;
- ❖ No contexto hacking, pode ser aplicada a diversas finalidades, como:
 - ❖ Network hacking;
 - ❖ Web hacking;
 - ❖ Binary Exploitation;
 - ❖ Heap Exploitation;
 - ❖ Entre outros...

IMPORTANTE:

- Python não é a melhor linguagem de programação, pois esta não existe.
- Não é a melhor indicada para iniciar na área de segurança, linguagens de baixo nível fornecem embasamento muito mais sólido sobre como tudo funciona.

Entendendo a Arquitetura

Conceitos básicos

Machine Code

Machine Code é um conjunto de instruções que a CPU (Central Process Unit) processa, estas instruções realizam operações lógicas e aritméticas, movem dados, entre outras funções. Todas estas instruções são representadas em formato hexadecimal.

```
48 31 d2
48 bb 2f 2f 62 69 6e
2f 73 68
48 c1 eb 08
53
48 89 e7
52
57
48 89 e6
b8 3b 00 00 00
0f 05
```


Assembly

Para facilitar para nós, humanos, a programação em linguagem de máquina, foi criado um código mnemônico chamado "Assembly (ASM)".

```
0000000000401000 <_start>:
401000: 48 31 d2          xor     rdx,rdx
401003: 48 bb 2f 2f 62 69 6e movabs  rbx,0x68732f6e69622f2f
40100a: 2f 73 68
40100d: 48 c1 eb 08      shr     rbx,0x8
401011: 53              push    rbx
401012: 48 89 e7         mov     rdi,rsi
401015: 52              push    rdx
401016: 57              push    rdi
401017: 48 89 e6         mov     rsi,rsi
40101a: b8 3b 00 00 00   mov     eax,0x3b
40101f: 0f 05           syscall
```

Registadores

Para otimizar as instruções no processamento, a CPU possui um conjunto de registradores. Estes registradores tem uma largura específica que muda de acordo com a arquitetura.

x86 = Processadores de 32 bits = 4 bytes de largura.

x64 = Processadores de 64 bits = 8 bytes de largura.

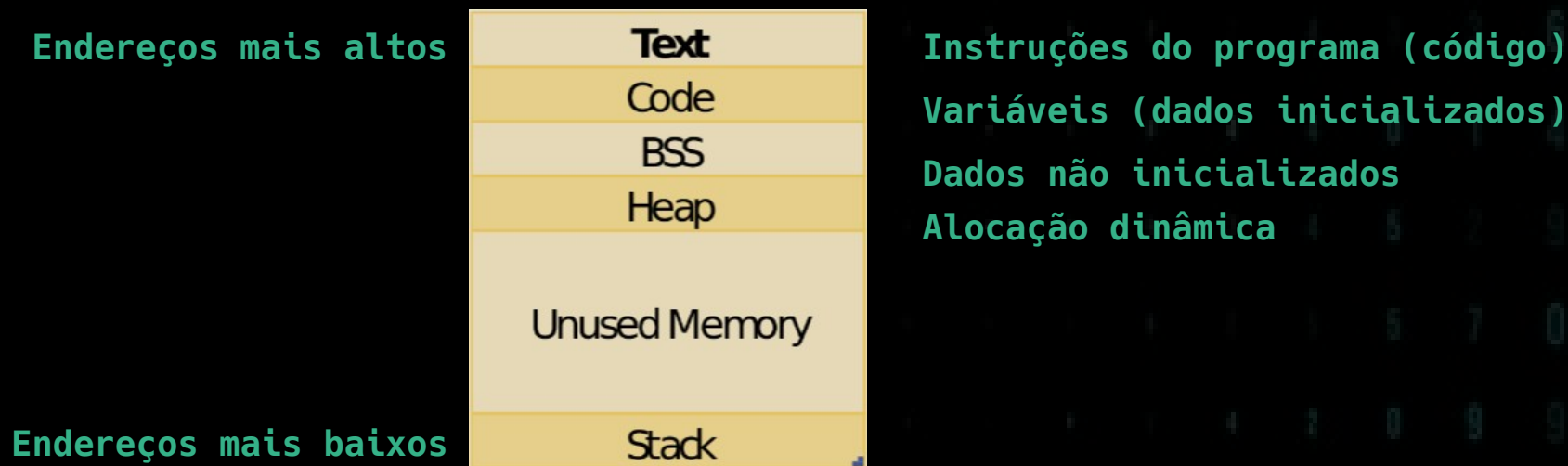
64 bits = 8 bytes	32 bits = 4 bytes	16 bits = 2 bytes	8 bits = 1 byte
RAX - Accumulator	EAX	AX	AH ~ AL
RBX - Base	EBX	BX	BH ~ BL
RCX - Counter	ECX	CX	CH ~ CL
RDX - Data	EDX	DX	DH ~ DL
RSI - Source Index	ESI	SI	
RDI - Destination Index	EDI	DI	
RSP - Stack Pointer	ESP	SP	
RBP - Base Pointer	EBP	BP	
RIP - Instruction Pointer	EIP	IP	
R8 ~ R15			

Registadores

Semânticamente, cada registrador tem sua própria função, porém, como consenso geral, à depender da utilização, os registradores **RAX**, **RBX**, **RCX**, e **RDY** são utilizados por propósito geral, pois podem ser repositórios para armazenar variáveis e informações. Já os registradores **RSI**, **RDI**, **RSP**, **RBP** e **RIP** tem a função de controlar e direcionar a execução do programa, e são exatamente estes registradores que iremos manipular na técnica de corrupção de memória.

Perspectiva de Execução

Durante a execução de um binário, na perspectiva da memória, existe uma arquitetura que organiza todos os *steps*.



Stack

Entre todas as etapas da execução do programa, a "Stack", ou pilha, é onde focaremos o ataque, pois ela é responsável por armazenar todos os dados, ou "buffer", que são imputados para o programa vindos de fora.

Basicamente, a Stack é usada para as seguintes funções:

- ❖ Armazenar argumentos de funções;
- ❖ Armazenar variáveis locais;
- ❖ Armazenar o estado do processador entre chamadas de função.

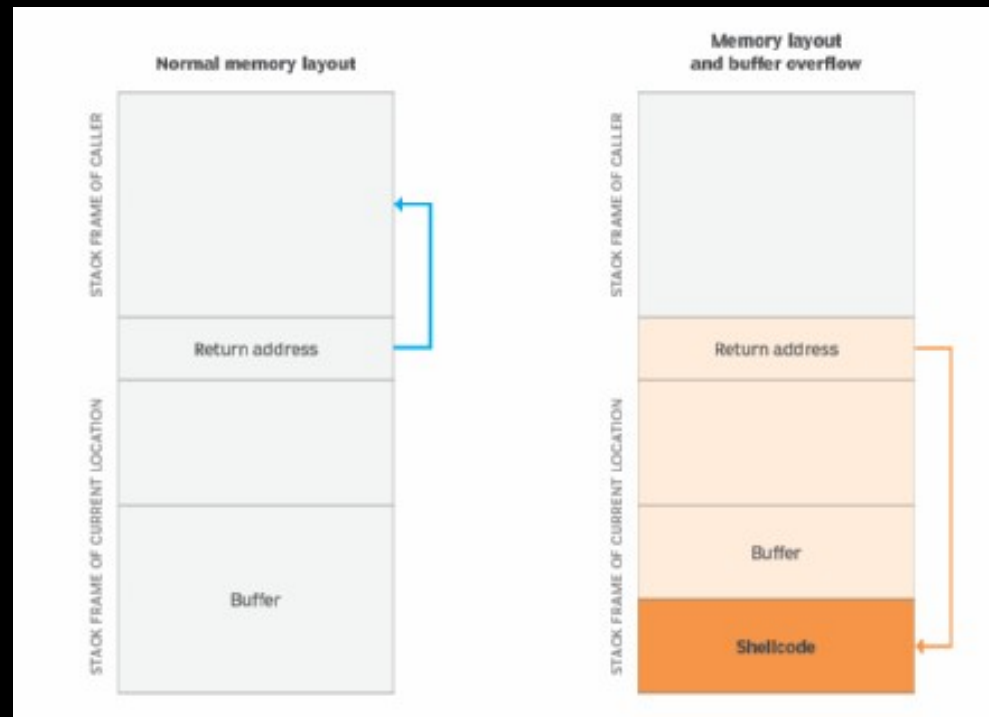
A Stack segue a ordem de execução "LIFO" (Last In First Out), onde o primeiro dado a entrar, é o último a sair. Seguindo esta ordem, o registrador RBP referencia a base da pilha, e o registrador RSP referencia o top da pilha.

Buffer Overflow

Conceitos básicos

Buffer Overflow

O buffer overflow, ou estouro de buffer, ocorre quando um programa recebe como entrada, um buffer maior do que o suportado, fazendo com que outros espaços da memória sejam sobrescritos. Quando um registrador que controla a execução, como o RIP, é sobrescrito por um valor inválido na memória, temos o buffer overflow que causa a “quebra” do programa, e ele pára sua execução. Porém, quando o sobreescrevemos com um endereço existente na memória do programa ou da Libc, conseguimos controlar o programa para executar funções maliciosas como execução de comandos arbitrários ou um reverse shell.



Mitigações e Meios de Proteção

Conceitos básicos

Principais Meios de Proteção

Existem alguns métodos e ferramentas utilizadas comumente para dificultar a manipulação e exploração de binários. Estes mecanismos não são infalíveis, mas, se utilizados em conjunto e implementadas de forma correta, podem aumentar muito a segurança de um binário. São elas:

NX

O bit No eXecute ou NX (também chamado de *Data Execution Prevention* ou DEP), marca certas áreas do programa, como a Stack, como não executáveis. Isso faz com que seja impossível executar um comando direto da Stack e força o uso de outras técnicas de exploração, como o ROP (Return Oriented Programming).

ASLR

A *Address Space Layout Randomization* se trata da randomização do endereço da memória onde o programa e as bibliotecas do programa estão. Em outras palavras, a cada execução do programa, todos os endereços de memória mudam. Desta forma, fica muito difícil durante a exploração, encontrar o endereço de alguma função sensível para utilizar de forma maliciosa.

PIE Protector

Muito parecido com o ASLR, o PIE Protector (*Position Independent Executables*) randomiza os endereços da Stack a cada execução, tornando impossível prever em quais endereços de memória, cada função do programa estará ao ser executado.

Principais Meios de Proteção

Stack Canaries

O Stack Canary é um valor randômico inserido na Stack toda vez que o programa é iniciado. Após o retorno de uma função, o programa faz a checagem do valor do canary, se for o mesmo, o programa continua, se for diferente, o programa pára sua execução. Em outras palavras, se sobrescrevermos o valor do Canary com o buffer overflow, e na checagem o valor não bater com o inserido pelo programa, nossa exploração irá falhar. É uma técnica bastante efetiva, uma vez que é praticamente impossível adivinharmos um valor randômico de 64 bits, porém existem formas de efetuar o bypass do Canary através de vazamento de endereços de memória e/ou bruteforce.

Shellcode em Assembly

O Linux Tem o que é Preciso

O próprio “man” do Linux, nos dá o que precisamos entender sobre a construção de um *shellcode*. O executermos:

```
$ man syscall
```

Podemos ver como o Linux trata as chamadas de sistema em diferentes arquiteturas.

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error	Notes
alpha	callsys	v0	v0	a4	a3	1, 6
arc	trap0	r8	r0	-	-	
arm/OABI	swi NR	-	r0	-	-	2
arm/EABI	swi 0x0	r7	r0	r1	-	
arm64	svc #0	w8	x0	x1	-	
blackfin	excpt 0x0	P0	R0	-	-	
i386	int \$0x80	eax	eax	edx	-	
ia64	break 0x100000	r15	r8	r9	r10	1, 6
m68k	trap #0	d0	d0	-	-	
microblaze	brki r14,8	r12	r3	-	-	
mips	syscall	v0	v0	v1	a3	1, 6
nios2	trap	r2	r2	-	r7	
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-	
powerpc	sc	r0	r3	-	r0	1
powerpc64	sc	r0	r3	-	cr0.SO	1
riscv	ecall	a7	a0	a1	-	
s390	svc 0	r1	r2	r3	-	3
s390x	svc 0	r1	r2	r3	-	3
superh	trap #0x17	r3	r0	r1	-	4, 6
sparc/32	t 0x10	g1	o0	o1	psr/csr	1, 6
sparc/64	t 0x6d	g1	o0	o1	psr/csr	1, 6
tile	swint1	R10	R00	-	R01	1
x86-64	syscall	rax	rax	rdx	-	5
x32	syscall	rax	rax	rdx	-	5
xtensa	syscall	a2	a2	-	-	

Arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	r0	r1	r2	r3	r4	r5	r6	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	1
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
powerpc64	r3	r4	r5	r6	r7	r8	-	
riscv	a0	a1	a2	a3	a4	a5	-	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
superh	r4	r5	r6	r7	r0	r1	r2	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
tile	R00	R01	R02	R03	R04	R05	-	
x86-64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	
xtensa	a6	a3	a4	a5	a8	a9	-	

As imagens nos mostram em qual registrador cada parâmetro deve ser inserido para uma *syscall* ser realizada em cada arquitetura.

O Linux Tem o que é Preciso

Para chamar uma função de sistema, precisamos de uma função que execute comandos do sistema operacional, uma das mais utilizadas em *shellcodes* é a **execve()**. Seu manual nos mostra como ela opera:

\$ **man** execve

```
SYNOPSIS
#include <unistd.h>

int execve(const char *pathname, char *const argv[],
           char *const envp[]);

DESCRIPTION
execve() executes the program referred to by pathname. This causes the
program that is currently being run by the calling process to be re-
placed with a new program, with newly initialized stack, heap, and (ini-
tialized and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a
line of the form:

    #!/interpreter [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its
command-line arguments. By convention, the first of these strings
(i.e., argv[0]) should contain the filename associated with the file be-
ing executed. The argv array must be terminated by a NULL pointer.
(Thus, in the new program, argv[argc] will be NULL.)
```

A função **execve** recebe três argumentos:

- ❖ **pathname** que recebe o endereço do comando a ser executado, neste caso será utilizado **"/bin/sh"**
- ❖ **argv[]** que é uma array de argumentos que deve ser iniciada com o path do programa e terminado em **NULL**
- ❖ **envp[]** que é uma array de chave=valor, para ser passada para o programa.

O Linux Tem o que é Preciso

Cada função dentro do Linux tem seu *opcode* em Assembly para ser utilizada em uma *syscall*. Podemos consultar o *opcode* da *execve* no seguinte diretório:

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep execve
```

```
$ cat /usr/include/x86_64-linux-gnu/asm/unistd_64.h | grep execve
#define __NR_execve 59
#define __NR_execveat 322
$ █
```

Com o valor encontrado de **59**, a estrutura do comando fica desta forma:

execve	pathname	argv[]
59	/bin/sh	['/bin/sh', 0]

Colocando Tudo em Ordem

Conhecendo os registradores e parâmetros que cada um recebe, e a arquitetura, podemos escrever o seguinte código.

```
global _start

section .text

_start:

    xor     rdx, rdx                ; Zerando o registrador RDX
    mov     qword rbx, '//bin/sh' ; Inserindo o comando //bin/sh em RBX
    shr     rbx, 8                  ; Shift Right de 8 bits em RBX para limpar a / extra
    push    rbx                    ; empurrando RBX para a Stack

    mov     rdi, rsp                ; Voltando o /bin/sh para RDI (argumento 1)
    push    rdx                    ; Enviando o NULL para a pilha
    push    rdi                    ; Enviando /bin/sh para a pilha
    mov     rsi, rsp                ; Movendo ["/bin/sh", 0] para RSI (argumento 2)
    mov     rax, 59                 ; Movendo para RAX o valor de execve
    syscall                        ; Chamando a função

$
```

Compilando o ELF

Com o código pronto, podemos “**assemblar**” para transformar o código mnemônico em linguagem de máquina e em seguida linka-lo para compilar em um executável com os seguintes comandos:

```
$ nasm -f elf64 shell.asm  
$ ld shell.o -o shell
```

E o resultado é um elf que executa “/bin/sh”.

```
leonardo@debian:/tmp$ nasm -f elf64 shell.asm  
leonardo@debian:/tmp$ ld shell.o -o shell  
leonardo@debian:/tmp$ ./shell  
$ whoami  
leonardo  
$ █
```


Gerando o Shellcode

Para que seja possível obter um *shellcode* utilizável em scripts e *exploits*, é preciso “disassemblar” o elf gerado e formatar a saída com um *Shell Script* conforme abaixo:

```
$ for i in $(objdump -d -M intel shell | grep '^ ' | cut -f2); do echo -n '\x'$i; done
```

E a resposta será o que precisamos.

```
leonardo@debian:/tmp$ for i in $(objdump -d -M intel shell | grep '^ ' | cut -f2); do echo  
-n '\x'$i; done && echo  
\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x52\x  
57\x48\x89\xe6\xb8\x3b\x00\x00\x00\x0f\x05
```

Vanilla Buffer Overflow

Vanilla BoF

Para fins de entendimento de como ocorre a corrupção de memória através do buffer overflow, esta primeira parte será realizada com um binário sem nenhuma proteção. Este experimento dará uma visão de como o programa é executado a nível de memória e como é possível executar um shellcode a partir da Stack. Para que o ASLR seja desativado, é preciso executar o comando abaixo:

```
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

HANDS ON!!!