SHELLCODING CHEAT SHEET

Introdução

Este é um guia rápido de consulta para criação de shellcode
Criado por Hélvio Junior (M4v3r1ck)

Registradores 32 bits

32 bits	16 bits	8 bits	
		High	Low
eax	ax	ah	al
ecx	cx	ch	cl
edx	dx	dh	dl
ebx	bx	bh	bl
esp	sp	--	spl
ebp	bp	--	bpl
esi	si	--	sil
edi	di	--	dil

Registradores 64 bits

64 bits	*32 bits*	*16 bits*	*8 bits*	
			High	Low
rax / r0	eax / r0d	ax / r0w	ah	al / r0b
rcx / r1	ecx / r1d	cx / r1w	ch	cl / r1b
rdx / r2	edx / r2d	dx / r2w	dh	dl / r2b
rbx / r3	ebx / r3d	bx / r3w	bh	bl / r3b
rsp / r4	esp / r4d	sp / r4w	--	spl / r4b
rbp / r5	ebp / r5d	bp / r5w	--	bpl / r5b
rsi / r6	esi / r6d	si / r6w	--	sil / r6b
rdi / r7	edi / r7d	di / r7w	--	dil / r7b
r8	r8d	r8w	--	r8b
r9	r9d	r9w	--	r9b
r10	r10d	r10w	--	r10b
r11	r11d	r11w	--	r11b
r12	r12d	r12w	--	r12b
r13	r13d	r13w	--	r13b
r14	r14d	r14w	--	r14b
r15	r15d	r15w	--	r15b

IP – Instruction Pointer

Endereço da próxima instrução a ser executada

ip 16 bits
eip 32 bits
rip 64 bits

SP – Stack Pointer

Armazena o endereço de memória que aponta para o topo da pilha (stack)

sp 16 bits
esp 32 bits
rsp 64 bits

Principais instruções Assembly

Instrução	Função
INT3	Software breakpoint
CALL	Chama função
CLD	Limpa a Direction Flag
DEC	Decrementa em 01 o registrador indicado
INC	Incrementa em 01 o registrador indicado
JMP	Salta para endereço especificado
LEA	Calcula endereço efetivo
MOV	Copia informação
NOP	Sem operação (Não faz nada)
POP	Remove do topo da pilha e insere em um registrador
PUSH	Insere dados no topo da pilha
RET	Sai de uma função
SHL	Deslocamento de bits a esquerda
SHR	Deslocamento de bits a direita
XOR	Operação XOR entre os registradores

Comandos GDB

Abrindo aplicação no GDB

gdb [path_da_aplicação]

Abrindo aplicação com seu coredump

gdb [path_da_aplicação] [path_core_dump]

HELP

(gdb) help [comando]
(gdb) help run

Rodando aplicação sem parâmetro

(gdb) run

Rodando aplicação com parâmetro

(gdb) run AAAAA

Rodando aplicação com parâmetro vindo de um comando externo

(gdb) run <(echo -n “AAAA”)
(gdb) run <(python -c 'print "CHALLENGE " + "A" * 500 ')

Rodando aplicação recebendo dados via stdin vindo de um comando externo

(gdb) run < <(echo -n “AAAA”)
(gdb) run < <(python -c 'print "CHALLENGE " + "A" * 500 ')

Adicionando breakpoint

(gdb) b *main
(gdb) b *0x01020304

Listando Breakpoints

(gdb) info breakpoints

Excluindo Breakpoints

(gdb) del 1
(gdb) del

Disasemble

(gdb) disassemble main
(gdb) disassemble/r main
(gdb) disassemble 0x000011e9,+100

Conteúdos dos registradores

(gdb) info registers
(gdb) info registers eax

Visualiza struct

(gdb) ptype struct [nome_da_struct]
(gdb) ptype struct sockaddr_in
(gdb) ptype/o struct [nome_da_struct]

Visualiza dados de uma posição de memória

(gdb) print &nome_variavel
(gdb) print 0x01020304

Visualiza dados de uma posição de memória com parse de struct

(gdb) print/x *(struct sockaddr_in *) &nome_variavel
(gdb) print/x *(struct sockaddr_in *) 0x01020304

Verificando proteções

(gdb) checksec

Shellcode Tester Linux

Instalando

git clone https://github.com/helviojunior/shellcodetester.git
cd shellcodetester/Linux
make

Assemblando (montando) utilizando shellcode tester Linux

shellcodetester arquivo.asm
shellcodetester arquivo.asm --break-point

ASM 32 bits para shellcode

```
[BITS 32]
global _start
section .text
_start:
    ; Instrucoes
```

ASM 64 bits para shellcode

Para 64 bits alterar a primeira linha para [BITS 64]

SHELLCODING CHEAT SHEET

Assemblando (montando) ASM para shellcode

Montando arquivo
nasm arquivo.asm -o arquivo.o

Opcodes em formato hexa
cat arquivo.o | msfvenom -p - -a x86 --platform win -e generic/none -f hex

Opcodes em formato python
cat arquivo.o | msfvenom -p - -a x86 --platform win -e generic/none -f python

ASM para geração ELF Linux
Em Linux não há alteração no ASM, sendo assim utilizar o mesmo dos exemplos acima.

ASM 32 bits para geração PE Windows

```
[BITS 32]
global WinMain
section .text
WinMain:
    ; Instrucoes
```

ASM 64 bits para geração PE Windows
Para 64 bits alterar a primeira linha para [BITS 64]

Assemblando (montando) ASM para execução

Linux 32 bits
nasm -f elf32 arquivo.asm -o arquivo.o
ld -o arquivo arquivo.o -m elf_i386

Linux 64 bits
nasm -f elf64 arquivo.asm -o arquivo.o
ld -o arquivo arquivo.o -m elf_x86_64

Windows 32 bits
nasm -f win32 arquivo.asm -o arquivo.o
gcc -o arquivo.exe arquivo.o

Windows 64 bits
nasm -f win64 arquivo.asm -o arquivo.o
gcc -o arquivo.exe arquivo.o

Texto em ordem reversa e print Hexa utilizando python
texto = "Treinamento Shellcoding\n"
texto[::-1]
len(texto[::-1])
texto[::-1].encode('hex')

ASM Tecnica JMP, CALL, POP

```
[BITS 32]
global _start
section .text
_start:
    jmp step1
step2:
    pop ecx ; Salva o endereço do texto em ECX
    nop
    ; Continua as instrucoes
step1:
    call step2
    db "Treinamento Shellcoding", 0x0a, 0x00
```

Passagem de parâmetros em chamada de função

Syscall Linux 32 bits

Registrador	Função
EAX	Numero do System Call
EBX	Primeiro parâmetro
ECX	Segundo parâmetro
EDX	Terceiro parâmetro
ESI	Quarto parâmetro
EDI	Quinto parâmetro

Pseudo code: Func1(ebx, ecx, edx, esi, edi)

Syscall Linux 64 bits

Registrador	Função
RAX	Número do System Call
RDI	Primeiro parâmetro
RSI	Segundo parâmetro
RDX	Terceiro parâmetro
R10	Quarto parâmetro
R8	Quinto parâmetro
R9	Sexto parâmetro

Pseudo code: Func1(rdi, rsi, rdx, r10, r8, r9)

Linux e Windows 32 bits (Stack)

Registrador	Função
ESP + 0x00	Primeiro parâmetro
ESP + 0x04	Segundo parâmetro
ESP + 0x08	Terceiro parâmetro
ESP + 0x0C	Quarto parâmetro
ESP + 0x10	Quinto parâmetro
ESP + 0x14	Sexto parâmetro
…	

Pseudo code: Func1(ESP, ESP + 0x04, ESP + 0x08, ...)

Linux e Windows 64 bits

Registrador	Função
RCX	Primeiro parâmetro
RDX	Segundo parâmetro
R8	Terceiro parâmetro
R9	Quarto parâmetro
ESP + 0x00	Quinto parâmetro
ESP + 0x04	Sexto parâmetro
…	

Pseudo code: func1(int a, int b, int c, int d, int e);
Onde: a em RCX, b em RDX, c em R8, d em R9, e adicionado na pilha

AMD64 Application Binary Interface (ABI) - Calling convention defaults

- Alinhamento
  - Pilha necessita estar alinhada a 16 bytes
- Passagem de parâmetros
  - RCX, RDX, R8, R9, o restante na pilha
- Shadow store
  - Criação de uma área vazia na pilha com a correspondência de um-para-um da quantidade de parâmetros da função que está sendo chamada
- Retorno de valores
  - Quando há retorno o valor e/ou ponteiro é salvo no registrador RAX

ASM 64 bits – instruções de exemplo

```
cld; Limpa a flag de direção

and rsp, 0xFFFFFFFFFFFFFFF0 ; Alinhamento em 16 bytes na pilha

xor eax,eax ; zera EAX
push eax    ; Coloca 1 shadow store na pilha
```

Simbolos de debug

Extraindo de uma aplicação
objcopy --only-keep-debug [arquivo_elf] simbolos.debug

Utilizando simbolos dentro do GDB
(gdb) symbol-file simbolos.debug