Brandenburg University of Technology Cottbus - Senftenberg

Introduction into Cyber Security,

Cottbus, Germany, December 19th, 2018

Introduction into Assembler and Shell-Coding

A. Panchenko, T. Ziemann

Research Group BTU Cottbus-Senftenberg,

Chair of IT Security

1. Introduction into Assembler

2. On the Way to Byte-Code

3. Literature / References

The Central Processing Unit (CPU)

Idea:

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions.[1]

[1] ref.: https://en.wikipedia.org/wiki/Central_processing_unit

The Central Processing Unit (CPU)

Idea:

A central processing unit (CPU) is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions.[1]

E.g.,

```
mov eax, $1    ; store 1 to eax
mov ebx, $12   ; store 12 to ebx
add eax, ebx   ; eax = eax + ebx
```

[1] ref.: https://en.wikipedia.org/wiki/Central_processing_unit

Processor Registers of x86 32 Bit CPU

eax	Accumulator Register
ebx	Base Register
ecx	Counter Register
edx	Data Register
edi	Destination Register
esi	Source Register
ebp	Base Pointer Register (Stack)
esp	Stack Pointer Register (Stack)
eip	Instruction Pointer

Tab. Registers of x86 32 Bit CPU[2]

[2] For more detailed information look at the “Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 2”

Some More Useful Instructions

xor <reg1> <reg2>	logic exclusive OR of <reg1> and <reg2>
mov <dst> <src>	move the value from <src> to <dst>
lea <dst> <src>	put the memory address of <src> into <dst>
push <val>	push the value <val> on top of the stack
pop <reg>	pop the top value of the stack to <reg>
call <addr>	call the function on address <addr>
ret	return statement
jmp <addr>	jump to address <addr>
int <id>	trap into kernel mode; use $id = 0x80$

Tab. Excerpt of the x86 32-Bit Instruction Set (NASM Syntax)[3]

[3] For more detailed information look at the "Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2"

Some More Useful Instructions

xor <reg1> <reg2>	logic exclusive OR of <reg1> and <reg2>
mov <dst> <src>	move the value from <src> to <dst>
lea <dst> <src>	put the memory address of <src> into <dst>
push <val>	push the value <val> on top of the stack
pop <reg>	pop the top value of the stack to <reg>
call <addr>	call the function on address <addr>
ret	return statement
jmp <addr>	jump to address <addr>
int <id>	trap into kernel mode; use $id = 0x80$

Tab. Excerpt of the x86 32-Bit Instruction Set (NASM Syntax)[3]

[3] For more detailed information look at the “Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 2”

Some System-Call Codes

$0x01$	void exit(int state)
$0x0b$	int execv(const char *path, char *const argv[])
$\vdots$	$\vdots$

Tab. Excerpt of the x86 32-Bit System-Call Codes[4]

[4]For more detailed information look at the “Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 2”

The Exit Program

```
; Simple Program to exit with a defined
; exit state
; ##################################################

global _start


section .text
_start:
        xor eax, eax              ; clear out eax register
        mov byte al, 0x01         ; Interupt ID for exit into eax
        xor ebx, ebx              ; return code is zero
        int 0x80                  ; trap into the kernel
```

Current Limitations

- limitations on count and size of the registers
- limitation on persistence of stored data

⇒ Solution: Memory architecture

Memory architecture in Operation Systems

Pyramid Structured Design

- Primary memory, e.g., RAM, Cache
- Secondary memory, e.g., Hard Drives, Solid State Disks, CD-ROM, etc.
- Tertiary memory (long time storage solutions)

Size of available memory per stage growth up while the speed slows down!

Memory architecture in Operation Systems

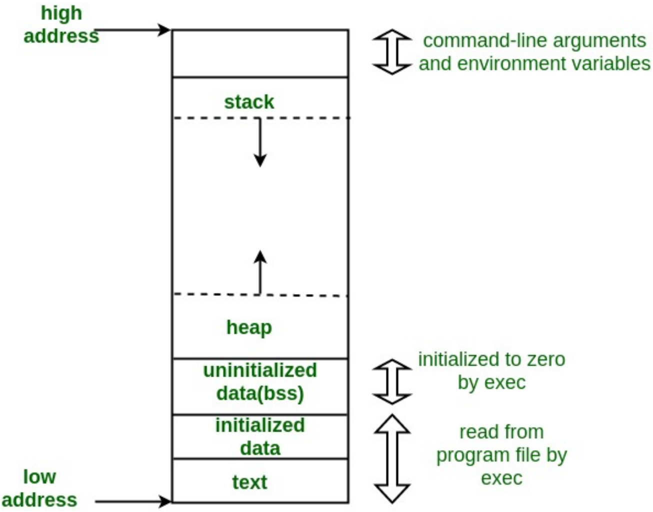
Pyramid Structured Design

- Primary memory, e.g., RAM, Cache
- Secondary memory, e.g., Hard Drives, Solid State Disks, CD-ROM, etc.
- Tertiary memory (long time storage solutions)

Size of available memory per stage growth up while the speed slows down!

Here, most important region of memory is the stack. Used to pass parameters to a function and store the local variables.

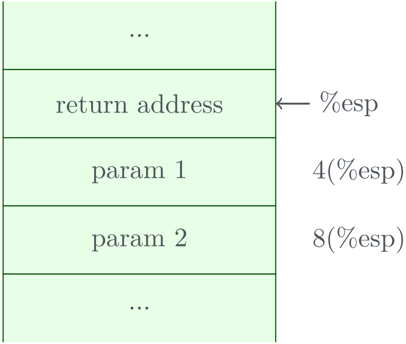
Memory Layout of C/ASM Programs



The Stack During a Function Call I

main(...) calls func(param1, param2)

- In the scope of main() parameters get pushed to the stack in reverse order
- address of the next instruction after the function func(param1, param2) is pushed to the stack (saved return address)
- main(...) triggers jump to address of func(...)



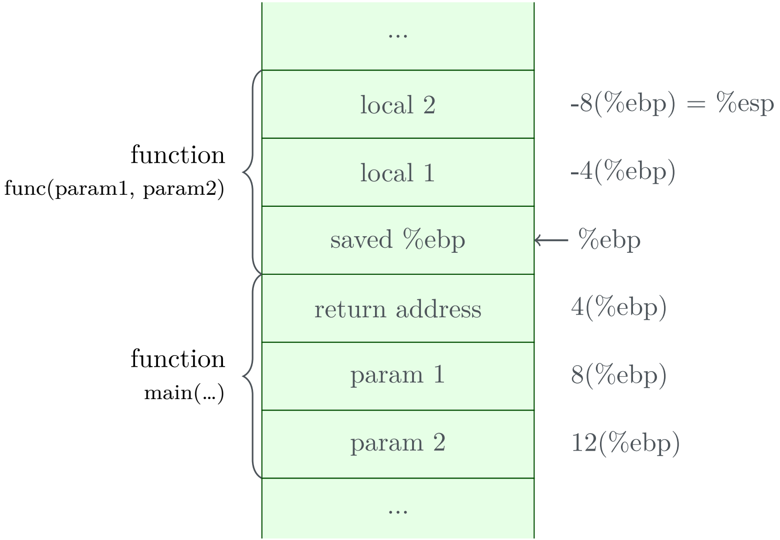
The Stack During a Function Call II

main(...) calls func(param1, param2)

- Now, in scope of func(...).
- func(...) backups the base pointer (ebp) by pushing its value on the stack.
- func(...) set the current stack pointer (esp) as its own base pointer (ebp).
- func(...) reserve space for additional local variables

...	
locals	-4(%ebp)
saved %ebp	← %ebp
return address	4(%ebp)
param 1	8(%ebp)
param 2	12(%ebp)
...	

The Complete Stack



Translated into Assembler

Function Prologue:

```
push ebp                    ; save the current base pointer
mov     ebp, esp            ; current top of stack (esp)
                            ; becomes base pointer
sub     esp, N              ; create space for local variables (N Bytes)
```

Translated into Assembler

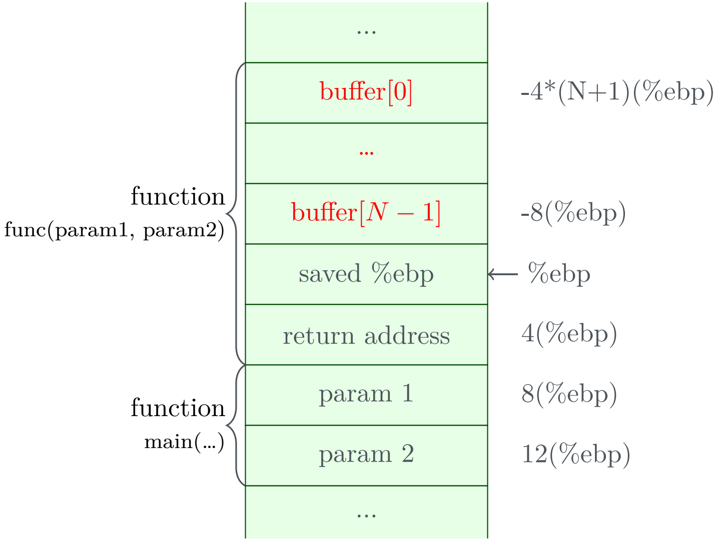
Function Prologue:

```
push ebp                    ; save the current base pointer
mov     ebp, esp            ; current top of stack (esp)
                            ; becomes base pointer
sub     esp, N              ; create space for local variables (N Bytes)
```

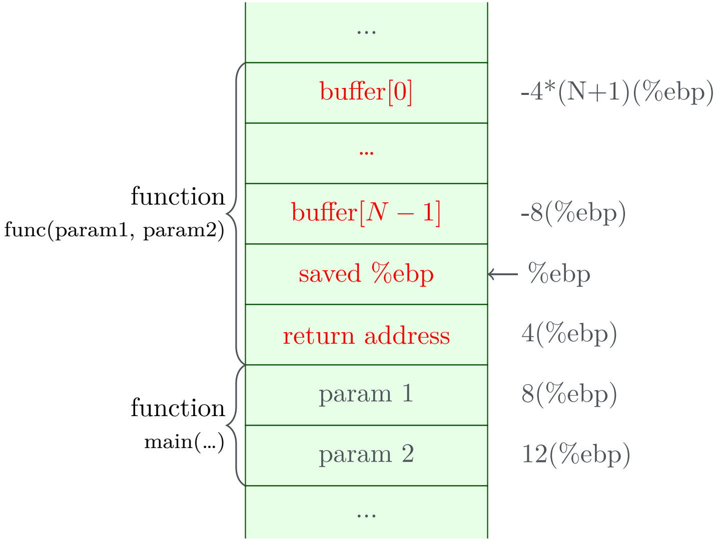
Function Epilogue:

```
mov     esp, ebp            ; delete stack frame (base pointer
                            ; becomes stack pointer)
pop     ebp                 ; restore saved base pointer
                            ; from callee
ret                         ; return aka pop saved return address
                            ; into %eip and jump to it
```

The Abstract Concept of Buffer Overflows



The Abstract Concept of Buffer Overflows



Steps of a Buffer Overflow

Recipe:

1. Find the necessary size of input to overwrite the saved return address
2. Prepare a buffer containing code to inject (so called Shell Code) for the input to the program
3. Overwrite the saved return address in such a way that the program flow continues on your (near to your) injected code

Steps of a Buffer Overflow

Recipe:

1. Find the necessary size of input to overwrite the saved return address
2. Prepare a buffer containing code to inject (so called Shell Code) for the input to the program
3. Overwrite the saved return address in such a way that the program flow continues on your (near to your) injected code

Therefore, we need to create program code compatible with the code already on the stack ! (Shell Code)

On the Way to Byte-Code I

```
; Simple Program to exit with a defined
; exit state
; ################################################

global _start


section .text
_start:
        xor eax, eax              ; clear out eax register
        mov byte al, 0x01         ; Interupt ID for exit into eax
        xor ebx, ebx              ; return code is zero
        int 0x80                  ; trap into the kernel
```

On the Way to Byte-Code I

```
; Simple Program to exit with a defined
; exit state
; ########################################

global _start


section .text
_start:
        xor eax, eax            ; clear out eax register
        mov byte al, 0x01       ; Interupt ID for exit into eax
        xor ebx, ebx            ; return code is zero
        int 0x80                ; trap into the kernel
```

1. Compile using nasm: nasm -f elf32 exit.asm
2. Link the object file using ld: ld -m elf_i386 -o exiter exit.o
3. Extract the binary by: objdump -d exiter

On the Way to Byte-Code II

1. Compile using nasm: nasm -f elf32 exit.asm
2. Link the object file using ld: ld -m elf_i386 -o exiter exit.o
3. Extract the binary by: objdump -d exiter

```

exiter:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       b0 01                   mov    $0x1,%al
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       cd 80                   int    $0x80
```

On the Way to Byte-Code II

1. Compile using nasm: nasm -f elf32 exit.asm
2. Link the object file using ld: ld -m elf_i386 -o exiter exit.o
3. Extract the binary by: objdump -d exiter

```

exiter:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       b0 01                   mov    $0x1,%al
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       cd 80                   int    $0x80
```

Shell-Code: “0x31 0xc0 0xb0 0x01 0x31 0xdb 0xcd 0x80”

Testing your Shell-Code

```c
// shellcode_launcher.c
// Simple program to test the generated Shellcode
//
// Compiler gcc-3-4
//
// ##############################################################


int main()
{
        // Shellcode saved as string of hex values
        char sc[] = "\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

        // execute shellcode
        int (*ret)() = (int (*)())sc;
        ret();
}
```

References

- Programming from the Ground Up by Jonathan Bartlett
  https://download-mirror.savannah.gnu.org/releases/pgubook/ProgrammingGroundUp-1-0-booksize.pdf
- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
- Shellcoding for Linux and Windows Tutorial
  http://www.vividmachines.com/shellcode/shellcode.html
- Smashing The Stack For Fun And Profit by Aleph One,
  http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
- ...

Thank you for your attention!