09/02/2025

Malicious Code Analysis

Fangtian Zhong
CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

Overview

01 Numbers, Strings, Arrays

02 Procedures, Recursion, Macro

03 File Management

04 Memory Management

Part One

01

Numbers, Strings, Arrays

Numbers

- Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data.
- When numbers are displayed on screen or entered from keyboard, they are in ASCII form.
- We have converted this input data in ASCII form to binary for arithmetic calculations and converted the result back to binary.

```
bits 64
default rel

section .data
    format db "%c", 0
    newline db 0x0A, 0x00
section .bss
    temrcx resq 1

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
    ;perform conversion
    mov rax, '3'
    sub rax, '0'

    mov rbx, '4'
    sub rbx, '0'
    add rax, rbx
    add rax, '0'

    ;print the result
    mov rcx, format
    mov rdx, rax
    call printf

    ; Exit the program
    xor eax, eax
    call ExitProcess
```

The ADD and SUB Instructions

- The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16-bit, 32-bit or 64-bit operands, respectively.
- The ADD and SUB instructions have the following syntax −
  - **ADD/SUB destination, source**
- The ADD/SUB instruction can take place between −
  - **Register to register**
  - **Memory to register**
  - **Register to memory**
  - **Register to constant data**
  - **Memory to constant data**

ASCII

ASCII Representation

- In ASCII representation, decimal numbers are stored as string of ASCII characters.
- For example, the decimal value 1234 is stored as −
  - **31 32 33 34H**
  - Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on.

Solution

```
bits 64
default rel

section .data
    format db "%c", 0
    newline db 0x0A, 0x00
section .bss
    result resq 1

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
; Subtract 6 from 'B' (ASCII value 66)
mov rax, 'B'
sub rax, 6


; Store the result in 'result' variable
mov qword [result], rax

;print the result
mov rcx, format
mov rdx, qword [result]
call printf

; Exit the program
xor rax, rax
call ExitProcess
```

Strings

- We have already used variable length strings in our previous examples. The variable length strings can have as many characters as required. Generally, we specify the length of the string by either of the two ways −
  - **Explicitly storing string length**
  - **Using a sentinel character**
- We can store the string length explicitly by using the $ location counter symbol that represents the current value of the location counter. In the following example −
  - **msg db 'Hello, world!',0xa ;our dear string**
  - **len equ $ - msg   ;length of our dear string**
- $ points to the byte after the last character of the string variable msg. Therefore, $-msg gives the length of the string. We can also write
  - **msg db 'Hello, world!',0xa ;our dear string**
  - **len equ 13   ;length of our dear string**

Strings

- Alternatively, … nel character to delimit a st… xplicitly. The sentinel chara… does not appear within
- For example …
  - **message d…**

```
bits 64
default rel

section .data
    format db "%d", 0
    msg db "Hello, world!", 0  ; Null-terminated string
    len equ $-msg-1      ; Length of the string (excluding null terminator)


section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ;print the len
    mov rcx, format
    mov rdx, len
    call printf

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

String Instructions

- Each string instruction may require a source operand, a destination operand or both. For 64-bit segments, string instructions use RSI and RDI registers to point to the source and destination operands, respectively.
- For 32-bit segments, however, the ESI and the EDI registers are used to point to the source and destination, respectively.

String Instructions

There are five basic instructions for processing strings. They are −

- **MOVS** − This instruction moves 1 Byte, Word or Doubleword of data from memory location to another.
- **LODS** − This instruction loads from memory. If the operand is of one byte, it is loaded into the AL register, if the operand is one word, it is loaded into the AX register and a doubleword is loaded into the EAX register.
- **STOS** − This instruction stores data from register (AL, AX, or EAX) to memory.
- **CMPS** − This instruction compares two data items in memory. Data could be of a byte size, word or doubleword.
- **SCAS** − This instruction compares the contents of a register (AL, AX or EAX) with the contents of an item in memory.

Examples

- These instructions use the ES:RDI and DS:RSI pair of registers, where RDI and RSI registers contain valid offset addresses that refers to bytes stored in memory. RSI is normally associated with DS (data segment) and RDI is always associated with ES (extra segment).

Versions of String Instructions

MOVSB	MOVSW	MOVSD	MOVSQ
LODSB	LODSW	LODSD	LODSQ
STOSB	STOSW	STOSD	STOSQ
CMPSB	CMPSW	CMPSD	CMPSQ
SCASB	SCASW	SCASD	SCASQ

Repetition Prefixes

- The REP prefix, when set before a string instruction, for example - REP MOVSB, causes repetition of the instruction based on a counter placed at the RCX register. REP executes the instruction, decreases RCX by 1, and checks whether RCX is zero. It repeats the instruction processing until RCX is zero.

- The Direction Flag (DF) determines the direction of the operation.
  - Use CLD (Clear Direction Flag, DF = 0) to make the operation left to right.
  - Use STD (Set Direction Flag, DF = 1) to make the operation right to left.

REP Prefix Variants

The REP prefix also has the following variations:

- **REP:** It is the unconditional repeat. It repeats the operation until CX is zero.
- **REPE or REPZ:** It is conditional repeat. It repeats the operation while the zero flag indicates equal/zero. It stops when the ZF indicates not equal/zero or when CX is zero.
- **REPNE or REPNZ:** It is also conditional repeat. It repeats the operation while the zero flag indicates not equal/zero. It stops when the ZF indicates equal/zero or when CX is decremented to zero.

Arrays

- We have already discussed that the data definition directives to the assembler are used for allocating storage for variables. The variable could also be initialized with hexadecimal, decimal or binary values.
- For example, we can define a word variable 'months' in either of the following way −
  - MONTHS DW 12
  - MONTHS DW 0CH
  - MONTHS DW 0110B
- The data definition directives can also be used for defining a one-dimensional array. Let us define a one-dimensional array of numbers.
  - NUMBERS DW 34, 45, 56, 67, 75, 89

Examples

- The times directive can also be used for multiple initializations to the same value. Using times, the inventory array can be defined as:
  - **inventory times 8 dw 0**

```
bits 64
default rel

section .data
    format db "%d", 0
    myArray dd 1, 2, 3, 4, 5    ; Define an array of 32-bit integers
section .bss
    sum resq 1


section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    ; Calculate the sum of the array
    mov rcx, 5          ; Set the loop counter to the number of elements in the array
    mov rsi, myArray    ; Load the base address of the array into ESI
    mov rdx, 0          ; Initialize the sum to 0

sum_loop:
    add rdx, [rsi]      ; Add the value at [ESI] to the sum in ECX
    add rsi, 4          ; Increment the pointer to the next element (since each dword is 4 bytes)
    loop sum_loop       ; Decrement the loop counter (EDX) and repeat until it reaches zero

    mov qword [sum], rdx; save the result to sum

    ;print sum
    mov rcx, format
    mov rdx, qword [sum]
    call printf

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

Part Two

02

Procedures, Recursion, Macro

Procedures

- Procedures are identified by a name. Following this name, the body of the procedure is described which performs a well-defined job. End of the procedure is indicated by a return statement.

- Following is the syntax to define a procedure −

```
proc_name:
   procedure body
   ...
   ret
```

call instruction

- The procedure is called from another function by using the call instruction. The call instruction should have the name of the called procedure as an argument as shown below −
  - call proc_name
- The called procedure returns the control to the calling procedure by using the ret instruction.

Examples

```
bits 64
default rel

section .data
    sumFormat db "The sum is: %c"

section .bss
    res resq 1


section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
```

```
    mov rcx, '4'
    sub rcx, '0'
    mov rdx, '5'
    sub rdx, '0'
    call sum  ;call sum procedure
    mov qword [res], rax

    mov rcx, sumFormat
    mov rdx, qword [res]
    call printf


    ; Exit the program
    xor rax, rax
    call ExitProcess
sum:
    mov rax, rcx
    add rax, rdx
    add rax, '0'
    ret
```

Stacks Data Structure

- A stack is an array-like data structure in the memory in which data can be stored and removed from a location called the 'top' of the stack. The data that needs to be stored is 'pushed' into the stack and data to be retrieved is 'popped' out from the stack. Stack is a LIFO data structure, i.e., the data stored first is retrieved last.
- Assembly language provides two instructions for stack operations: PUSH and POP. These instructions have syntaxes like −
  - PUSH   operand
  - POP   address/register
- The memory space reserved in the stack segment is used for implementing stack. The registers SS and RSP (ESP or SP) are used for implementing the stack. The top of the stack, the last data item inserted into the stack is pointed to by the SS:RSP register, where the SS register points to the beginning of the stack segment and the RSP (ESP or SP) gives the offset into the stack segment.

Stack Characteristics

- The stack implementation has the following characteristics −
  - Only words or doublewords could be saved into the stack, not a byte.
  - The stack grows in the reverse direction, i.e., toward the lower memory address.
  - The top of the stack points to the last item inserted in the stack; it points to the lower byte of the last word inserted.
- As we discussed about storing the values of the registers in the stack before using them for some use; it can be done in following way −

```
; Save the RAX and RBX registers in the stack
PUSH    RAX
PUSH    RBX
; Use the registers for other purpose
MOV     RAX, VALUE1
MOV     RBX, VALUE2
...
MOV     VALUE1, RAX
MOV     VALUE2, RBX

; Restore the original values
POP     RBX
POP     RAX
```

Macro

- Writing a macro is another way of ensuring modular programming in assembly language.

  A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.

- The Syntax for macro definition −
  - %macro macro_name number_of_params
  - <macro body>
  - %endmacro

  Where, number_of_params specifies the number parameters, macro_name specifies the name of the macro.

- The macro is invoked by using the macro name along with the necessary parameters. When you need to use some sequence of instructions many times in a program, you can put those instructions in a macro and use it instead of writing the instructions all the time.

Macros

- For example, a very common need for programs is to write a string of characters in the screen.
- For displaying a string of characters, you need the following sequence of instructions −

```
lea     rcx, [msg]
call    printf
```

```
bits 64
default rel

%macro PrintString 1
    lea rcx, [%1]
    call printf
%endmacro

section .data
    msg1 db "Hello, programmers!",0xA,0xD
    msg2 db "Welcome to the world of,", 0xA,0xD
    msg3 db "Windows assembly programming! "

section .text
    extern printf
    global main
    extern _CRT_INIT
    extern ExitProcess

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT

    PrintString msg1
    PrintString msg2
    PrintString msg3

    ; Exit the program
    xor rax, rax
    call ExitProcess
```

Part Three

03

File Management

File Management System

- File management can be accomplished using the WinAPI functions provided by the operating system. These functions allow you to perform various operations such as creating, opening, reading, writing, and closing files.There are three standard file streams −

  - Standard input (stdin),
  - Standard output (stdout), and
  - Standard error (stderr).

Create or Open a File:

- To cre… e Create… file name, desire… It returns a hand… ons.

```
main:

    push    rbp
    mov    rbp, rsp


    call    _CRT_INIT                 ; Needed since our entry point is not _DllMainCRTStartup.

    ; Additional arguments pushed onto stack
    xor    eax, eax
    push rax
    push FILE_ATTRIBUTE_NORMAL
    push CREATE_ALWAYS

    ; First 4 arguments go in registers, rest are pushed onto the stack
    xor    r9, r9
    mov    r8d, FILE_SHARE_READ
    mov    rdx, GENERIC_WRITE|GENERIC_READ
    lea    rcx, [default_filename]

    sub rsp, 32
    call    CreateFileA
    add rsp, 32

    cmp    rax, INVALID_HANDLE_VALUE
    je    .error_creating_file
```

Create or Open a File:

```
extern HeapAlloc
extern HeapFree
extern ReadFile
extern WriteFile

extern printf

global main

main:
.dwCreationDisposition equ 32
.dwFlagsAndAttributes equ 36
.hTemplateFile equ 40

    push    rbp
    mov     rbp, rsp
    sub     rsp, 64

    call    _CRT_INIT; Needed since our entry point is not _DllMainCRTStartup. See https://msdn.microsoft.com/en-us/library/708by912.aspx

    ; Additional arguments pushed onto stack
    xor     eax, eax
    mov     qword [rsp + .hTemplateFile], rax
    mov     dword [rsp + .dwFlagsAndAttributes], FILE_ATTRIBUTE_NORMAL
    mov     dword [rsp + .dwCreationDisposition], CREATE_ALWAYS

    ; First 4 arguments go in registers, rest are pushed onto the stack
    xor     r9, r9
    mov     r8d, FILE_SHARE_READ
    mov     rdx, GENERIC_WRITE|GENERIC_READ
    lea     rcx, [default_filename]
    call    CreateFileA

    cmp     eax, INVALID_HANDLE_VALUE
    je     .error_creating_file

.error_creating_file:
    mov     eax, 1                  ; Oh no, we failed.
    jmp     .quit_program

.quit_program:
    call    ExitProcess
```

Write

- To write data to a file, you can use the WriteFile function. Similar to ReadFile, you need to provide the file handle, a buffer containing the data to write, the number of bytes to write, and other required parameters. The function writes the specified number of bytes from the buffer to the file.

```
BOOL WriteFile(
  [in]                HANDLE       hFile,
  [in]                LPCVOID      lpBuffer,
  [in]                DWORD        nNumberOfBytesToWrite,
  [out, optional]     LPDWORD      lpNumberOfBytesWritten,
  [in, out, optional] LPOVERLAPPED lpOverlapped
);
```

Write

```
    ; NOTE: Clear the stack of the original variables; this is necessary since
    ; launching from cmder can make it possible to have stack corruption otherwise.

.lpNumberOfBytesWritten equ 52
.lpOverlapped equu 0
.fileHandle equ 40

    mov     rbp, rsp
    sub     rsp, 64


    mov     rcx, rax ; File handle is first argument to WriteFile
    xor     rax, rax
    mov     dword [rsp + .lpOverlapped], eax
    lea     qword r9, [rsp + .lpNumberOfBytesWritten]
    xor     r8, r8
    mov     dword r8d, default_text_length
    lea     rdx, [default_text]
    call     WriteFile

    cmp     eax, 0
    je     .error_writing_to_file
```

Close

- After you finish working with a file, it's important to close it using the CloseHandle function. This function takes the file handle as a parameter and releases any resources associated with the file.

```
mov     qword [rsp + .fileHandle], rax; save handle for closing file
mov     rcx, rax ; File handle is first argument to WriteFile
xor     rax, rax
mov     dword [rsp + .lpOverlapped], eax
lea     qword r9, [rsp + .lpNumberOfBytesWritten]
xor     r8, r8
mov     dword r8d, default_text_length
lea     rdx, [default_text]
call     WriteFile

cmp     rax, 0
je     .error_writing_to_file

mov     ecx, dword [rsp + .fileHandle]
call     CloseHandle

cmp     rax, 0
je     .error_closing_file
```


```
mov     qword [rsp + .fileHandle], rax; save handle for closing file
mov     rcx, rax ; File handle is first argument to WriteFile
xor     rax, rax
mov     dword [rsp + .lpOverlapped], eax
lea     qword r9, [rsp + .lpNumberOfBytesWritten]
xor     r8, r8
mov     dword r8d, default_text_length
lea     rdx, [default_text]
call     WriteFile

cmp     rax, 0
je     .error_writing_to_file

mov     ecx, dword [rsp + .fileHandle]
call     CloseHandle

cmp     rax, 0
je     .error_closing_file
```

Read

- To read data from a file, you can use the ReadFile function. You need to provide the file handle, a buffer to store the read data, the number of bytes to read, and other necessary parameters. The function reads the specified number of bytes from the file and stores them in the provided buffer.

```
xor     rax, rax
mov     qword [rsp + .hTemplateFile], rax
mov     dword [rsp + .dwFlagsAndAttributes], FILE_ATTRIBUTE_NORMAL
mov     dword [rsp + .dwCreationDisposition], OPEN_EXISTING
xor     r9, r9
mov     r8d, FILE_SHARE_READ
mov     rdx, GENERIC_READ
lea     rcx, [default_filename]
call     CreateFileA

cmp     eax, INVALID_HANDLE_VALUE
je     .error_creating_file
```

Part Four

04

Memory Management

Memory Management System

- Memory management is primarily handled by the operating system. However, as a developer, you can interact with the memory management system using various WinAPI functions to allocate, deallocate, and manipulate memory.

Allocate Memory

- To allocate memory dynamically, you can use the HeapAlloc function. This function allows you to specify the size of the memory block, the desired allocation type, and other parameters. It returns a pointer to the allocated memory block.

```
DECLSPEC_ALLOCATOR LPVOID HeapAlloc(
  [in] HANDLE hHeap,
  [in] DWORD  dwFlags,
  [in] SIZE_T dwBytes
);
```

Allocate Memory

```
;return heap handle
mov     dword [rsp + .fileHandle], eax

call     GetProcessHeap

cmp     rax, 0
je     .error_getting_heap

;allocate heap space
xor     r8, r8
mov     r8, 256
mov     rdx, qword HEAP_ZERO_MEMORY
mov     rcx, rax
call     HeapAlloc

cmp     rax, 0
je     .error_allocating_memory

mov     rbp, rsp
sub     rsp, 64
```

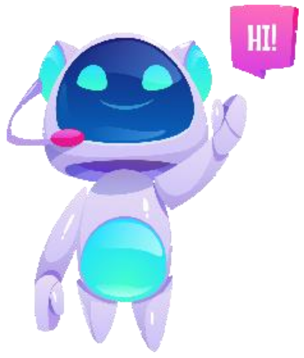
Read or Write Memory

- Once you ... n or write to it using ... ssembly, such as m...

```
.lpOverlapped2 equ 0
.lpNumberOfBytesRead equ 32
.lpBuffer equ 40


    mov     [rsp + .lpBuffer], rax; Store pointer to allocated memory block from HeapAlloc

    xor     rax, rax
    mov     [rsp + .lpOverlapped2], eax
    xor     r9, r9
    lea     r9, [rsp + .lpNumberOfBytesRead]
    xor     r8, r8
    mov     r8d, dword [rbp + .lpNumberOfBytesWritten]
    mov     rdx, qword [rsp + .lpBuffer]
    mov     rcx, qword [rbp + .fileHandle]
    call     ReadFile

    cmp     eax, 0
    je     .error_reading_file

    mov     rcx, qword [rbp + .fileHandle]
    call     CloseHandle

    cmp     eax, 0
    je     .error_closing_file

    mov     rdx, qword [rsp + .lpBuffer]
    lea     rcx, [final_printout]
    call     printf

    xor     eax, eax              ; return 0
    jmp     .quit_program
```

Deallocate Memory

- To release the previously allocated memory, you can use the HeapFreefunction. This function takes the pointer to the memory block and frees the associated memory. It also allows you to specify the desired release type and other parameters.



THE END

Fangtian Zhong

CSCI 591

Gianforte School of Computing
Norm Asbjornson College of Engineering
E-mail: fangtian.zhong@montana.edu

09/02/2025