Digital Forensics
Lecture 3 - Reverse Engineering

Low-Level Software

Akbar S. Namin
Texas Tech University
Spring 2018

Reverse Engineering – High-Level Software

- Low-level aspects of software are often the only ones visible for RE
- High-Level Perspectives
  - Modules
    - The largest building block for a program
    - Binary files that contain isolated areas of a program's executable
    - Two basic types of modules
      - Static libraries
        - » Represent a feature or an area of functionalities in the program
        - » An external, third-party library that adds certain functionality
      - Dynamic libraries
        - » Dynamic Link Libraries (DLLs)
        - » Similar to static libraries, except that they are not embedded into the program
        - » They remain in a separate file
        - » It allows for upgrading individual components in a program without updating the entire program

Reverse Engineering – High-Level Software

- Low-level aspects of software are often the only ones visible for RE
- High-Level Perspectives
  - Common Code Constructs
    - Two basic code-level constructs:
      - Procedures
        - » A piece of well-defined code that can be invoked by other areas in the program
      - Objects
        - » Dividing a program into objects

Reverse Engineering – High-Level Software

- Low-level aspects of software are often the only ones visible for RE
- High-Level Perspectives
  - Data Management
    - Variables
      - The key to managing and storing data
    - User-Defined Data Structures
      - Simple constructs to represent a group of data fields, each with its own type
      - Related fields
    - Lists: a group of data items that share the same data type
      - Arrays
      - Linked lists
      - Trees

Reverse Engineering – Low-Level Software

- Low-level aspects of software are often the only ones visible for RE
- Low-level data management
- High-level programming languages hide details of data management
- Even ANSI C (a relatively low-level language) hides significant data management details

Reverse Engineering – Low-Level Software

```
int Multiply(int x, int y)
{
int z;
z = x * y;
return z;
}
```

- Example: The Multiply program
- A low-level representation should:
  - Store machine state prior to executing code
  - Allocate memory for z
  - Load parameters x and y from memory into internal processor memory (registers)
  - Multiply x by y and store the result in a register
  - Optionally copy the multiplication result back into the memory area previously allocated for z
  - Restore machine state stored earlier
  - Return to called and send back z as the return value

Reverse Engineering – Low-Level Software

- Registers (internal memory)
  - Used by microprocessors to avoid having to access the RAM for every instruction
  - Small chunks of internal memory that resides within the processor
  - Can be accessed very easily with no performance penalty
  - Assembly language code usually uses them (complexity)
  - In the example:
    - x and y cannot be directly multiplied from memory
    - The code must first load one of them into a register and then multiply that register by the value that exist in RAM
  - Compilers also use them for storing (caching) frequently used values
  - When reversing, the values loaded into each register must be detected

Reverse Engineering – Low-Level Software

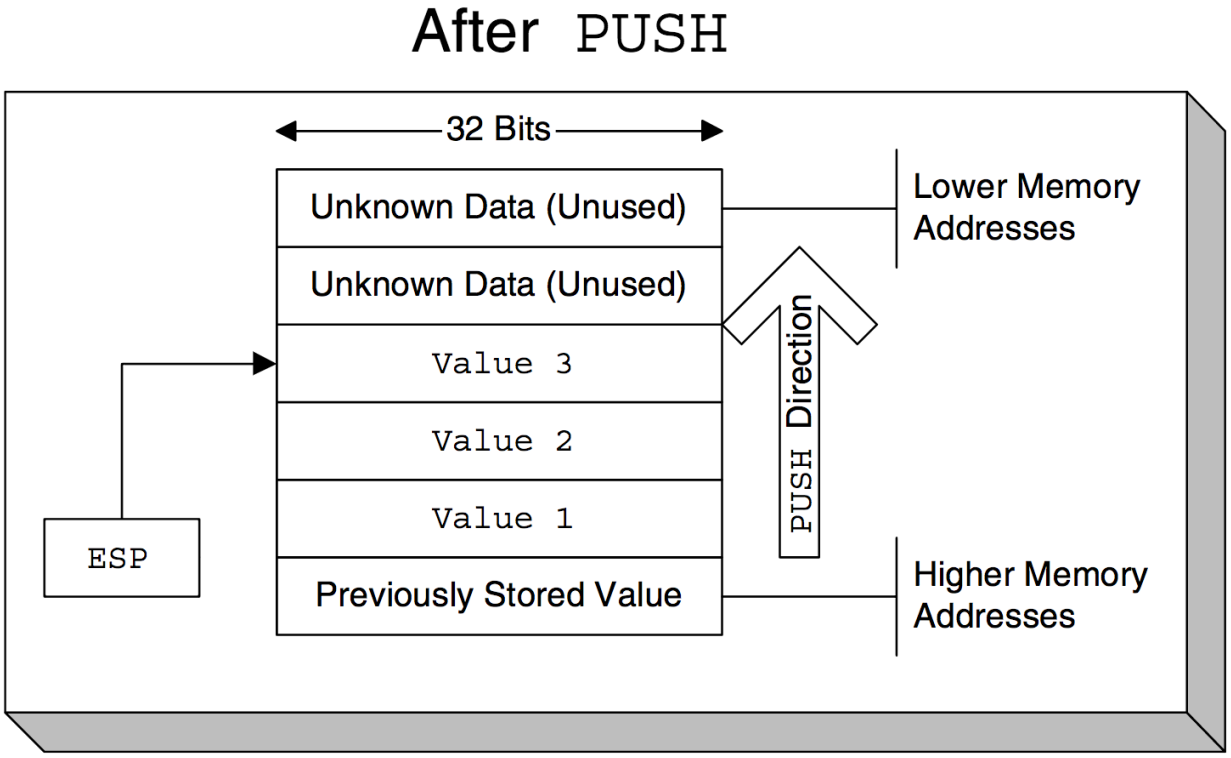
- The stack
  - An area in program memory is used for short-term storage of information by the CPU and the program
  - Registers are used for storing the most immediate data; stack is used for storing slightly longer-term data
  - Stacks reside in RAM
  - Memory for stacks is allocated from the top down
    - Stack grows backward towards the lower addresses

Reverse Engineering – Low-Level Software

- The stack: Push

Code Executed:

```
PUSH Value 1
PUSH Value 2
PUSH Value 3
```

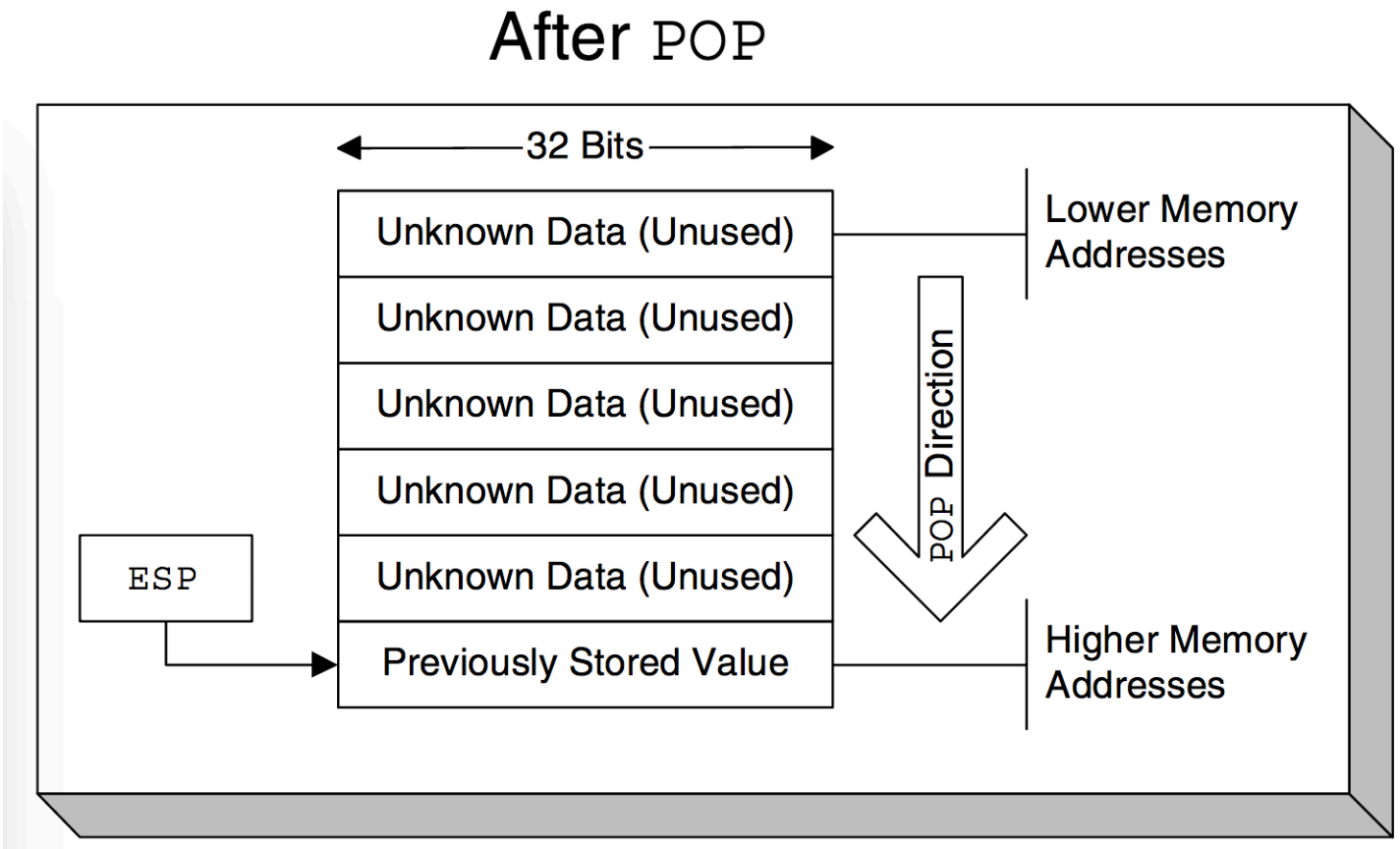


Reverse Engineering – Low-Level Software

- The stack: Pop

Code Executed:

```
POP  EAX
POP  EBX
POP  ECX
```



Reverse Engineering – Low-Level Software

- Stack can be used for:
  - Translating saved register values
    - Used when a procedure is called that needs to make use of certain registers
    - The procedure might need to preserve the values of registers to ensure it does not corrupt any registers by its callers
  - Local variables
    - Used for storing local variables that must be stored in RAM
    - E.g., when we want to call a function and have it write a value into a local variable defined in the current function
  - Function parameters and return addresses
    - Used for implementing function calls
    - The caller passes parameters to the callee
    - Stack is used for storing both parameters and the instruction pointer for each procedure call

Reverse Engineering – Low-Level Software

- Heaps
  - Allows the dynamic allocation of variable-sized blocks of memory in runtime
  - A program requests a block of a certain size and receives a pointer to the newly allocated block
  - The variable-sized objects are used by the program for objects that are too big to be placed on the stack
  - For reversing: locating heaps in memory and properly identifying heap allocation and freeing routines can be helpful
  - Eg.
    - A call to a heap allocation routine, we can follow the flow of the procedure's return value throughout the program and see what is done with the allocated block

Reverse Engineering – Low-Level Software

- Executable Data Sections
  - Used for storing application data
  - In high-level languages, this area contains either global variables ot pre-initialized data
  - Pre-initialized data:
    - Any kind of constant, hard-coded information included with the program
    - Any kind of hard-coded string inside a program
    - E.g., “char szWelcome = “This string will be stored in the executable’s pre-initialized data section”;”
      - Causing the compiler to store the string in the executable’s pre-initialized data section, regardless of where in the code szWelcome is declared
  - To access this string, the compiler emits a hard-coded address that points to the string
  - When reversing, it is identifiable, because hard-coded memory addresses are rarely used for anything other than pointing to the executable’s data section

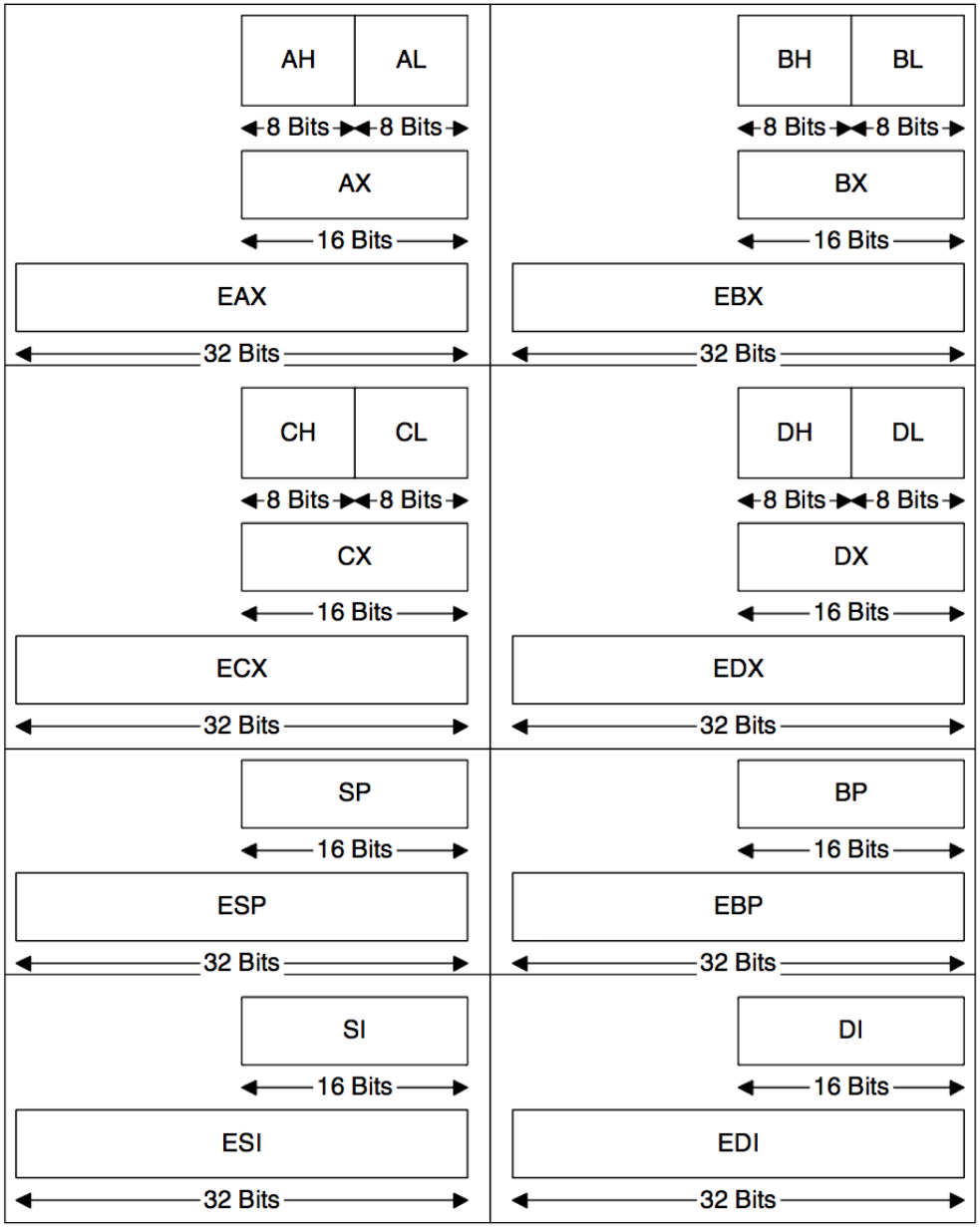
Reverse Engineering – Low-Level Software

- Based on Intel's 32-bit architecture (x86 CPUs)
- Registers
  - Eight generic registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP
  - E stands for "Extended"

`EAX, EBX, EDX`	These are all generic registers that can be used for any integer, Boolean, logical, or memory operation.
`ECX`	Generic, sometimes used as a counter by repetitive instructions that require counting.
`ESI/EDI`	Generic, frequently used as source/destination pointers in instructions that copy memory (`SI` stands for Source Index, and `DI` stands for Destination Index).
`EBP`	Can be used as a generic register, but is mostly used as the stack base pointer. Using a base pointer in combination with the stack pointer creates a *stack frame*. A stack frame can be defined as the current function's stack zone, which resides between the stack pointer (`ESP`) and the base pointer (`EBP`). The base pointer usually points to the stack position right after the return address for the current function. Stack frames are used for gaining quick and convenient access to both local variables and to the parameters passed to the current function.
`ESP`	This is the CPUs stack pointer. The stack pointer stores the *current position* in the stack, so that anything pushed to the stack gets pushed below this address, and this register is updated accordingly.

Reverse Engineering – Low-Level Software

- General purpose registers



Reverse Engineering – Low-Level Software

- Flags
  - Special register called EFLAGS
  - Contains all kinds of status and system flags
  - System Flags: Used for managing the various processor modes and states
  - Status Flags: Used by the processor for recording its current logical state
    - updated by many logical and integer instructions in order to record the outcome of their actions
  - A basic tool for creating conditional code
    - There are arithmetic instructions to test operands for certain conditions
    - There are instructions that read these flags and perform different operations depending on their values
    - E.g., Jcc (Conditional Jump) instructions (like GOTO)

Reverse Engineering – Low-Level Software

- Flags, Example of conditional jump
  - Consider a variable "bSuccess"
  - And a code that test whether it is false or not
    - if (bSuccess == FALSE) return 0;
  - Its assemble language version?
    - We must test the value of "bSuccess"
      - Loaded into a register first
    - Set some flags that record whtether it is zero ot not
    - Invoke a conditional branch instruction that tests the necessary flags
    - Branch if they indicate that the operand handled in the most recent instruction was zero (using the Zero Flag (ZF))
    - Otherwise, the processor will proceed to execute the instruction that follows the branch instruction

Reverse Engineering – Low-Level Software

- Arithmetic instructions
INSTRUCTION	DESCRIPTION
`ADD Operand1, Operand2`	Adds two signed or unsigned integers. The result is typically stored in `Operand1`.
`SUB Operand1, Operand2`	Subtracts the value at `Operand2` from the value at `Operand1`. The result is typically stored in `Operand1`. This instruction works for both signed and unsigned operands.
`MUL Operand`	Multiplies the unsigned operand by `EAX` and stores the result in a 64-bit value in `EDX:EAX`. `EDX:EAX` means that the low (least significant) 32 bits are stored in `EAX` and the high (most significant) 32 bits are stored in `EDX`. This is a common arrangement in IA-32 instructions.
`DIV Operand`	Divides the unsigned 64-bit value stored in `EDX:EAX` by the unsigned operand. Stores the quotient in `EAX` and the remainder in `EDX`.
`IMUL Operand`	Multiplies the signed operand by `EAX` and stores the result in a 64-bit value in `EDX:EAX`.
`IDIV Operand`	Divides the signed 64-bit value stored in `EDX:EAX` by the signed operand. Stores the quotient in `EAX` and the remainder in `EDX`.

Reverse Engineering – Low-Level Software

- Comparing operands
  - "CMP operand1, Operand2"
  - The result of the comparison is recorded in the processor's flags
  - In fact: "CMP" subtracts Operand2 from Operand1 and discards the result, while setting all of the relevant flags
  - If the result of the subtraction is zero, the Zero Flag (ZF) is set.

Reverse Engineering – Low-Level Software

- Conditional branches
  - "Jcc TargetCodeAddress"
  - Conditionally branch to a specific address, based on certain conditions

- Function calls
  - "CALL FunctionAddress"
  - Call a function and the RET instruction returns to the caller
  - The CALL instruction pushes the current instruction pointer onto the stack
  - When a function completes, it invokes the RET instruction
  - RET pops the instruction pointer pushed to the stack by CALL and resumes execution from that address
  -