App RE

Saturday, December 29, 2018 9:26 PM

<u>2 main kinds of reverse engineering:</u>

1. **Static**
2. **Dynamic**

Static	Process of disassembling a compiled executable into native machine code ▪ Using that code to understand how the executable works
Dynamic	Executing an application/using tools like debuggers/function monitors ▪ Inspect the application's runtime operation

<u>Compilers, Interpreters and Assemblers:</u>

- The way a program executes determines how it's reverse engineered

Interpreted Languages: Ex. Python/Ruby/Scripting langs

- Commonly run from short scripts written as text files
- Dynamic/speed up dev time
- Interpreters execute programs more slowly than code that has been converted to machine code

<u>Compiled Languages:</u> Use a compiler to parse source code/generate machine code

- Typically generating intermediate lang first
- For native code generation: Assembly language specific to CPU

Static vs. Dynamic Linking

Linking: Process that uses a linker program after compilation

- Takes app-specific machine code generated by compiler along w/external libs/embeds everything into exe

Static	Process produces single/self-contained exe that doesn't depend on original libs ▪ OS-specific implementations could change
Dynamic	Instead of embedding machine code in final exe: ▪ Compiler stores only a ref to dynamic lib/required function ▪ OS must resolve linked references when app runs

x86 Architecture: Originally released by Intel 1978 w/8086 CPU

- Support over the years to 16/32/64-bit operations

ISA: Instruction Set Architecture

- Defines how machine code works/interacts w/CPU-rest of computer
- Defines set of instructions avail to a program:
  - Each individual machine lang instruction represented by mnemonic instruction
- **Mnemonics**: Name each instr/determine how params/operands represented

Common x86 Instruction Mnemonics

Instruction	Description
MOV dest, src	Moves value from source to dest
ADD dest, value	Adds int value to dest
SUB dest, value	Subs int value from dest
CALL address	Calls subroutine at specified addr
JMP address	Jumps unconditionally to specified addr
RET	Returns from previous subroutine
RETN size	Returns from previous subroutine/increments stack by size
Jcc addr	Jumps to specified addr if condition indicated by cc true
PUSH value	Pushes value onto stack/decrements stack pointer
POP dest	Pops top of stack into dest/increments stack pointer
CMP valuea, valueb	Compares valuea-b/sets appropriate flags
TEST valuea, valueb	Bitwise AND on valuea-b/sets appropriate flags

AND dest, value	Bitwise AND on dest w/value
OR dest, value	Bitwise OR on dest w/value
XOR dest, value	Bitwise Exclusive OR on dest w/value
SHL dest, N	Shifts dest to left by N bits [left being higher bits]
SHR dest, N	Shifts dest to right by N bits [right being lower bits]
INC dest	Increments dest by 1
DEC dest	Decrements dest by 1

Mnemonic instructions take 1 of 3 forms depending on how many ops instruction takes

Intel Mnemonic Forms

Num of operands	Form	Example
0	NAME	POP, RET
1	NAME input	PUSH 1; CALL func
2	NAME output, input	MOV EAX, EBX; ADD EDI, 1

2 common ways to represent x86 instructions:

1. **Intel**
2. **AT&T syntax**

<u>Example:</u> Add 1 to value in EAX register: **Intel: ADD EAX, 1** || **AT&T: add1 $1, %eax**

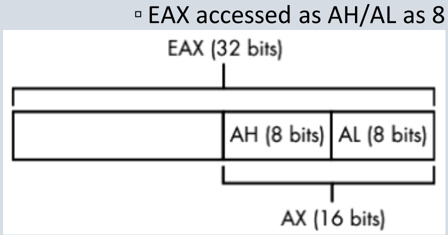
CPU Registers:

- The CPU has a number of registers for fast temp storage of current state of execution
- x86: Each registered referred to by 2/3-char label

Split into 4 main categories:

1. **General purpose**
2. **Memory index**
3. **Control**
4. **Selector**

<u>**Register Categories**</u>

<table>
<tr><td>General Purpose</td><td>EAX, EBX, ECX, EDX: 32 bit: Can access in 16/8 bit versions
▪ Temp stores for nonspecific values of computation
▪ Results of addition/subtraction
<u>Examples</u>:
▫ EAX accessed as AX for 16 bit
▫ EAX accessed as AH/AL as 8 bit

</td></tr>
<tr><td>Memory Index</td><td>ESI, EDI, ESP, EBP, EIP: Mostly general purpose except ESP/EIP
ESP register: Used by PUSH/POP
▪ Subroutine calls indicate current mem loc of base of stack
▪ Can be used for purposes other than indexing into stack
▫ <u>Not good</u>: Mem corruption/unexpected behavior
▫ Some instructions implicitly rely on value of register
EIP: Can't be directly accessed as general purpose register
▪ Indicates next addr in mem where instruction will be read from
▪ Only way to change value of EIP is by using a control instruction
CALL, JMP, RET
EFLAGS: Boolean flags that indicate results of instruction execution
▪ Whether last op resulted in value 0
▪ Implement conditional branches on x86 processor
▪ Also impt sys flags: Whether interrupts enabled</td></tr>
</table>

Selector	**CS, DS, ES, FS, GS, SS:** Addr mem locs: Specific block you can read/write ▪ Real mem addr used in read/write value looked in internal CPU table ▪ Usually OS specific ops	

Mem: Accessed using little endian byte order: LSB stored at lowest mem addr

- x86 arch doesn't req its mem ops to be aligned
- All reads/writes to main mem on aligned processor arch must be aligned to size of op
  - Example: To read 32-bit value: Would have to read from mem addr multiple of 4
    - Archs like SPARC: Reading unaligned addr would generate error
- x86 permits you to read from or write to any mem addr regardless

MOV EAX, [ESI + EDI * 8 + 0x50]   ; Read 32-bit value from memory address

Important EFLAGS Status Flags

Bit	Name	Description
0	Carry flag	Whether carry bit generated from last op
2	Parity flag	Parity of LSB of last op
6	Zero flag	Whether last op had 0 as result: Comparison ops
7	Sign flag	Sign of last op: MSB of result
11	Overflow flag	Whether last op overflowed

Program/Control Flow: How a program determines which instructions to execute

x86: 3 main types of program flow instructions

1. **Subroutine calling**
2. **Conditional branches**
3. **Unconditional branches**

Subroutine calling	**Redirects flow of program to subroutine** **Subroutine**: Specified sequence of instructions: Achieved w/**CALL** instruction ▪ Changes EIP to loc of subroutine CALL ▪ CALL places mem addr of next instr onto current stack ▪ Tells program flow where to ret after performed subroutine ▪ Return performed using RET ▫ Changes EIP to top address in stack
Conditional branches	**Allow code to make decisions on prior ops** Example: CMP compares 2 operands/calcs values for **EFLAGS** register Does this by: ▫ Subtracting 1 value from other ▫ Setting **EFLAGS** as appropriate ▫ Discards result **TEST instruction:** Does the same: Performs **AND** op instead of sub ▪ After EFLAGS value calc: Conditional branch can be exe ▪ Addr it jumps to depends on state of **EFLAGS** Example: **JZ** will conditionally jump if 0 flag set ▫ Happens if 2 values equal: Otherwise instruction is no-op **EFLAGS:** Can also be set by arithmetic/other instructions ▪ **SHL instruction**: Shifts value of dest by certain num of bits from low to high ▪ Implemented through **JMP**: Just jumps unconditionally to a dest addr

Exe File Formats: Modern exe fmts include:

- Mem allocation for exe instructions/data
- Support for dynamic linking of external libs
- Support for crypto sigs to validate source of exe
- Maintenance of debug info to link exe code to original src code for debugging
- Reference to addr in exe file where code begins executing (start addr)
  - Necessary: Program's start addr might not be 1st instruction in exe file

Windows: PE: Portable Executable fmt: Typically .exe extension

- **.dll Dynamic libraries**
- Doesn't actually need these extensions for a new process to work correctly: Convenience

Unix-like sys: ELF: Executable Linking Format: Primary exe fmt

MacOS: Mach-O fmt
Sections: Mem sections probably most impt info stored in an exe
<u>All nontrivial exe's have at least 3 sections</u>

1. **Code:** Contains native machine code for exe
2. **Data:** Contains initialized data that can be read/written during exe
3. **BSS:** Special section to contain uninitialized data:

Every section contains 4 basic pieces of info:

- Txt name
- Size/loc of data for section contained in exe file
- Size/addr in mem where data should be loaded
- Mem protection flags: Indicate whether section can be written/exe when loaded into mem

<u>Processes/Threads:</u>

Process	Acts as a container for an instance of a running exe ▪ Stores all private mem the instance needs to operate ▫ Isolates it from other instances of the same exe ▫ Also sec boundary: Runs under a particular usr of OS
Thread	Allows OS to rapidly switch bet multiple processes: ▪ Makes it seem like they're all running at the same time ▪ Defines current state of execution ▪ Own block of mem for a stack/somewhere to store its state when OS stops it
Multitasking	To switch bet processes: OS must interrupt CPU/store current processes state ▪ Restore an alternate process's state ▪ When CPU resumes: Running another process

OS Networking Int: Needs to provide a way for apps to int w/network
<u>Berkeley sockets model:</u> Most common network API: Dev at Uni. of CA, in 70's for BSD: All UNIX-like sys:

- Built-in support for Berkeley sockets

Winsock: Windows library provides similar programming int
<u>Creating a Simple TCP Client Connection to a Server:</u>

```
int port = 12345;
const char* ip ="1.2.3.4";
sockaddr_in addr = {0};

int s = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family = PF_INET;
addr.sin_port = htons(port);
inet_pton(AF_INET, ip, &addr.sin_addr);

if(connect(s, (sockaddr*) &addr, sizeof(addr)) == 0)
{
    char buf[1024];
    int len = recv(s, buf, sizeof(buf), 0);

    send(s, buf, len, 0);
}
close(s);
```

1. **<u>Creates new socket:</u> AF_INET** indicates we want to use IPv4/6
   - 2nd param **SOCK_STREAM** indicates to use streaming connection: **TCP**
   - To create UDP socket: **SOCK_DGRAM**
2. Call to **inet_pton** converts str representation of IP to 32-bit int
   - Convert value host-byte-order (x86 Little Endian) to network-byte-order (BE: Big Endian)
   - Applies to IP also: 1.2.3.4 will become int 0x01020304 when sorted in BE fmt
3. Issue call to connect to destination addr
   - Main point of failure: OS has to make outbound call to dest addr to see if anything listening
   - **New socket connection established?** Program can read/write data to socket as if it were a file via recv/send sys calls

<u>Creating a Client Connection to a TCP Server</u>

```
sockaddr_in bind_addr = {0};

int s = socket(AD_INET, SOCK_STREAM, 0);

bind_addr.sin_family = AD_INET;
bind_addr.sin_port = htons(12345);
inet_pton("0.0.0.0", &bind_addr.sin.addr);

bind(s, (sockaddr*)&bind_addr, sizeof(bind_addr));
listen(s, 10);

sockaddr_in client_addr;
int socksize = sizeof(client_addr);
int newsock = accept(s, (sockaddr*)&client_addr, &socksize);
```

1. Bind socket to an addr on local network int
2. Ensure server socket will be accessible from outside current sys assuming no fw
3. Listing asks network int to listen for new incoming connections/calls accept
4. Returns next new connection: New socket can be read/written w/recv/send calls

ABI: Application Binary Interface:

- Int defined by OS to describe conventions of how app calls API function
- <u>Most languages/OS's pass params left to right:</u>
  - Leftmost param in original source code placed at lowest stack addr
  - If params built by pushing them to a stack: Last param pushed first
- How return value provided to function's caller when API call is complete
  - **x86:** As long as value less than/equal to 32 bits: Passed back in EAX register
  - If value bet 32-64 bits: Passed back in combo of EAX/EDX

EAX/EDX: Scratch registers in ABI

- Register values aren't preserved across function calls
- <u>When calling a function:</u> Caller can't rely on any value stored in these registers to still exist
- Model of designating registers as scratch done for pragmatic reasons
  - Allows functions to spend less time/mem saving registers
  - ABI specifies an exact list of which registers must be saved into a loc on stack by called function

Saved Register List

Register	ABI usage	Saved?
EAX	Pass return value of function	No
EBX	General purpose	Yes
ECX	Local loops/counters: Sometimes pass object ptrs in C++	No
EDX	Extended return values	No
EDI	General purpose	Yes
ESI	General purpose	Yes
EBP	Ptr to base of current valid stack frame	Yes
ESP	Ptr to base of stack	Yes

Static RE: Process of dissecting exe to determine what it does:

- Ideally: Reverse compilation process to original source code: Usually difficult: More common to disassemble

objdump: Prints disassembled output to console/file

IDA Pro: Hex Rays: Go to tool for static RE

debug symbols	Info about original source code line associated w/an instruction in mem ▪ Type info for functions/vars: Devs rarely leave debug symbols intentionally
PDB	**Program Database File:** All debug info stored in file ▪ Separation of debug symbols from exe: Easy to distribute w/out debug info ▪ Rarely distributed w/exe's in closed-source software <u>MS Windows exception to this:</u> Releases public symbols for most exe's: Including kernel to aid debugging
dSYM	**Debugging Symbols Package**: Created alongside exe rather than single PDB file ▪ Separate macOS package dir/rarely distributed w/commercial apps
magic	**Numbers defined by an algorithm that are chosen for particular mathematical properties**

constants	▪ Tells if encryption alg compiled into exe Example: MD5 hashing alg: **void md5_init(md5_context *ctx)** **{** **ctx -> state[0] = 0x67452301;** **ctx -> state[1] = 0xEFCDAB89;** **ctx -> state[2] = 0x98BADCFE;** **ctx -> state[3] = 0x10325476;** **}** **IDA: Search > Immediate value > OK** Better tools can do searches for you **PEiD**: Determines whether a Windows PE file is packed w/known packing tool like UPX ▪ Plugins which can detect potential encryption algs/indicate where in exe referenced ▪ **Plugins > Krypto Analyzer**

Dynamic RE: Inspecting the op of a running exe: Useful when analyzing complex functionality

- Example: Custom crypto/compression routines
- Can step through one instruction at a time: Let's you test understanding of cold (inject test inputs)

Common way: Debugger halts running ap at specific points/inspects data values

- **IDA: Debugger > Process > options > Parameters** txt || To stop debugging running process: **CTRL F2**

Setting Breakpoints: Places of interest in disassembly: **F2**

- When program tries to exe instruction at breakpoint: Debugger should stop/give access to current program state

Debugger Windows: Default: IDA Pro debugger 3 impt win when hits breakpoint

1. **EIP Window**
2. **ESP Window**
3. **State of General Purpose Registers**

EIP Window	Displays disassembly view based on instruction in EIP register currently being executed ▪ Much like disassembly window ▪ Hovering mouse over register: Quick preview of value
ESP Window	Reflects current location of ESP register: Points to base of current thread's stack ▪ Can ID params being passed to function calls/value of local vars
General Purpose	Stores current values of various program states (loop counters/mem addr) ▪ Mem addr: Window provides way to navigate to mem view window ▫ Click arrow next to each addr to navigate from last active mem to value ▫ Create new mem window: Right click array > Jump in new window ▫ Lists condition flags from EFLAGS register on right side

Where to set breakpoints? send/recv functions/crypto functions

RE Managed Languages: Not apps distributed as native executables

- Apps written in managed languages: .NET/Java: Compile to an intermediate machine lang
- Commonly designed to be CPU/OS agnostic
- When app exe: VM/runtime executes code

.NET: Intermediate language called CIL: Common Intermediate Language

Java: Byte code: Substantial amts of metadata: Names of classes/internal-external-facing method names

- Output of managed languages fairly predictable: Ideal for decompiling

.NET app relies on:

1. **CLR: Common Runtime Language Runtime**
2. **BCL: Base Class Library**

Assemblies	.NET uses exe/dll fmts as convenient containers for CIL code ▪ Contain 1/more classes enums/structs ▪ Each type referred to by a name: Namespace/short name ▪ Namespace reduces likelihood of conflicting names but useful for categorization

ILSpy: Tools like Reflector/ILSpy can decompile CIL data into C#/CB source and display original Java Apps: Differ from .NET apps bc **Java compiles don't merge all types into single file**

- Compiles each source code file into single class file w/.class extension
- Java apps packaged into JAR: Java Archive format (just a zip w/addl files to support Java)

JD-GUI: Decompilation: Same as ILSpy for .NET

Obfuscation: Tackling Tips

- **External lib types/methods [core class] can't be obfuscated:** Calls to socket API's must exist in app if doing any networking
- **.NET/Java easy to load/exe dynamically:** Write simple test harness to load obfuscated app: Run str/code decryption routines
- **Use dynamic RE as much as possible to inspect types at runtime**: Determine what used for

Resources: OpenRCE http://www.openrce.org || **ELF:** http://refspecs.linuxbase.org/elf/elf.pdf