Software security, secure programming

Lecture 4: How to look at the binary level

Master M2 Cybersecurity & MoSiG

Academic Year 2019 - 2020

Reminder

So far, we saw that:

- Unsecure softwares are (almost) everywhere . . .
- Programming languages (often) contribute to produce unsecure software
- Looking at the source-level may not be enough
  $\rightarrow$ the binary level of the code matters, e.g.:
  - undefined behaviors + optimisations
  - memory layout
  - cache accesses
  - etc.

$\Rightarrow$ analysing the code level semantics is not sufficient for vulnerability detection and analysis . . .

$\Rightarrow$ **need tool and technique to understand the executable code**

Outline

Reverse Engineering

Inspecting the binary code

Some demos . . .

Software = several knowledge/information levels

- specifications, algorithms, data structures
- source code structure
  objects, variables, types, functions, control and data flows (CFGs and DFGs)
- assembly and
- binary code (executable)
  stack layout, optimizations,

Rk: $\exists$ sometimes a **bytecode** level (Java, LLVM, etc.)

Reverse-engineering in practice:

- disassembling: binary $\rightarrow$ assembly level
- de-compiling: binary $\rightarrow$ source level
- source level $\rightarrow$ model level ...

Reverse engineering a software - When ? (2)

- when the source code not/no longer available
  - updating, maintaining legacy code
  - analyzing COST, extern libraries
  - security (vulnerability, malware)

- when source code not sufficient enough

  what you see is not what you execute [T. Reps]

  optimization, memory layout, undefined behavior, protections, etc.

Rks

- source and/or binary code may be obfuscated ...
- in some situations need to consider only memory dumps

Memory layout at runtime (simplified)

Executable code = (binary) file produced by the compiler
$\rightarrow$ need to be loaded in memory to be executed (using a *loader*)

However:

- no abolute addresses are stored in the executable code
  $\rightarrow$ decided at "load time"
- not all the executable code is stored in the executable file
  (e.g., dynamic libraries)
- data memory can be dynamically allocated
- data can become code (and conversely . . .)
- etc.

The executable file should contain all the information . . .

$\exists$ standards executable formats: ELF (Linux), PE (Windows), etc.

- header
- sections: text, initialized/unitialized data, symbol tables, relocation tables, etc.

Rks: stripped (no symbol table) vs verbose (debug info) executables . . .

x86 (32) assembly language in one slide

Registers:

- stack pointer (ESP), frame pointer (EBP), program counter (EIP)
- general purpose: EAX, EBX, ECX, EDX, ESI, EDI
- flags

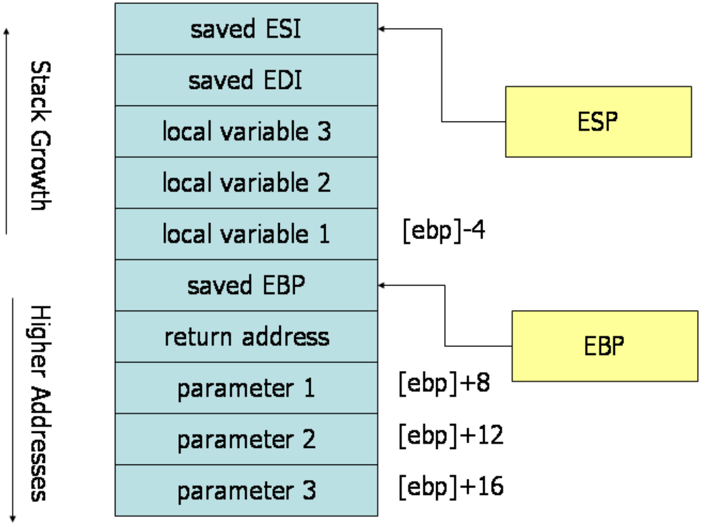
Instructions:

- data transfer (MOV), arithmetic (ADD, etc.)
- logic (AND, TEST, etc.)
- control transfer (JUMP, CALL, RET, etc)

Adressing modes:

- register: mov eax, ebx
- immediate: mov eax, 1
- direct memory: mov eax, [esp+12]

Stack layout for the x86 32-bits architecture



http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

ABI (Application Binary Interface)

to “standardize” how processor resources should be used
$\Rightarrow$ required to ensure compatibilities at binary level

- sizes, layouts, and alignments of basic data types
- **calling conventions**
  argument & return value passing, saved registers, etc.
- system calls to the operating system
- the binary format of object files, program libraries, etc.

	Cleans Stack	Arguments	Arg Ordering
cdecl	Caller	On the Stack	Right-to-left
fastcall	Callee	ECX,EDX, then stack	Left-to-Right
stdcall	Callee	On the Stack	Left-to-Right
VC++ thiscall	Callee	EDX (this), then stack	Right-to-left
GCC thiscall	Caller	On the Stack (this pointer first)	Right-to-left

Figure: some calling conventions

Outline

Reverse Engineering

Inspecting the binary code

Some demos . . .

Understanding and analysing binary code ?

```
01010100 01101000
01101001 01101110
01101011 00100000
01100100 01101001
01100110 01100110
01100101 01110010
01100101 01101110
01110100 00101110
```

```
00000000    push   ebp
00000001    mov    ebp, esp
00000003    movzx  ecx, [ebp+arg_0]
00000007    pop    ebp
00000008    movzx  dx, cl
0000000C    lea    eax, [edx+edx]
0000000F    add    eax, edx
00000011    shl    eax, 2
00000014    add    eax, edx
00000016    shr    eax, 8
00000019    sub    cl, al
0000001B    shr    cl, 1
0000001D    add    al, cl
0000001F    shr    al, 5
00000022    movzx  eax, al
00000025    retn
```

Disassembling !

statically: disassemble the **whole** file content without executing it . . .

dynamically: disassemble the **current** instruction path under execution/emulation . . .

Static disassembly

Recovering assembly-level code

- a non trivial task $\rightarrow$ static disassembling of x86 code undecidable (dynamic jumps, variable-length instructions, etc.)
  main issue: distinguishing code vs data ...
- several existing strategies (linear sweep, recursive disassembly, etc.)
- produce assembly-level IR instead of native assembly code
  $\rightarrow$ simpler language (a few instruction opcodes), explicit semantics (no side-effects), share analysis back-ends

Some existing tools

- IDA Pro
  a well-known commercial disassembler, # useful features
- On Linux plateforms (for ELF formats):
  - `objdump` (`-S` for code disassembling)
  - `readelf`

Static disassembly (cont'd)

See other slides available on the web page . . .

Dynamic disassembly

Main advantage: disassembling process **guided by** the execution

- ensures that instructions only are disassembled
- the whole execution context is available (registers, flags, addresses, etc.)
- dynamic jump destinations are resolved
- dymanic libraries are handled
- etc.

However:

- only a (small) part of the executable is disassembled
- need some suitable execution plateform, e.g.:
  - emulation environment
  - binary level code instrumentation
  - (scriptable) debugger
  - etc.

Outline