Software security, secure programming

Reverse-engineering from binary code

Master M2 Cybersecurity

Academic Year 2022 - 2023
Outline

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools . . .
Software = several knowledge/information levels

- (formal) models: overall architecture, component behaviors
- specifications, algorithms, abstract data structures
- source code
  objects, variables, types, functions, control and data flows
- possible intermediate representations: Java bytecode, LLVM IR, etc.
- assembly
- binary code (relocatable / shared object / executable)

Some reverse-engineering settings:

- source level → model level . . .
- de-compiling: binary → source level
- disassembling: binary → assembly level
- etc.
Why and when bothering with binary code? (1)

→ when the source code is not/no longer available

▶ updating/maintaining legacy code

▶ “off-the-shell” components (COST), external libraries

▶ dynamically loaded code (applets, plugins, mobile apps)

▶ pieces of assembly code in the source

▶ suspicious files (malware, etc.)
Why and when bothering with binary code? (2)

→ when the source code is not sufficient


▶ untrusted compilation chain

▶ low-level bugs, at the HW/SW interface

▶ security analysis
   going beyond standard programming language semantics
   (optimization, memory layout, undefined behavior, protections, etc.)

Beware! Reverse-engineering is restricted by the law
(“Intellectual Property”, e.g. Art. L122-6-1 du Code de la Propriété Intellectuelle)
Outline

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools . . .
Example 1: Java ByteCode (stack machine)

```java
public static int main() {
    int x, r;
    x=42 ; r=1 ;
    while (x>0) {
        r = r*x;
        x = x-1;
    } ;
    return r ;
}
```

```java
public static int main(java.lang.String[]);
Code:
0: bipush 42
2: istore_1
3: iconst_1
4: istore_2
5: iload_1
6: ifle 20
9: iload_2
10: iload_1
11: imul
12: istore_2
13: iload_1
14: iconst_1
15: isub
16: istore_1
17: goto 5
20: iload_2
21: ireturn
```

1use javap -c to produce the bytecode
Example 2: LLVM IR (register based machine)

```c
int main() {
  int x, r;
  x=42; r=1;
  while (x>0) {
    r = r*x;
    x = x-1;
  }
  return r;
}
```

CFG for 'main' function
```c
%0:
  %1 = alloca i32, align 4
  %x = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 0, i32* %1
  store i32 42, i32* %x, align 4
  store i32 1, i32* %r, align 4
  br label %2

%2:
  %3 = load i32* %x, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

T
%5:
  %6 = load i32* %r, align 4
  %7 = load i32* %x, align 4
  %8 = mul nsw i32 %6, %7
  store i32 %8, i32* %r, align 4
  %9 = load i32* %x, align 4
  %10 = sub nsw i32 %9, 1
  store i32 %10, i32* %x, align 4
  br label %2

F
%11:
  %12 = load i32* %r, align 4
  ret i32 %12
```

CFG for 'main' function
**Example 3: assembly code (x86-64)**

```c
int main() {
    int x, r;
    x=42; r=1;
    while (x>0) {
        r = r*x;
        x = x-1;
    }
    return r;
}
```

```assembly
main:
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-4], 42
mov DWORD PTR [rbp-8], 1
jmp .L2
.L3:
    mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-4]
mov DWORD PTR [rbp-8], eax
sub DWORD PTR [rbp-4], 1
.L2:
    cmp DWORD PTR [rbp-4], 0
    jg .L3
mov eax, DWORD PTR [rbp-8]
ret
```

---

²see https://godbolt.org/
Memory layout at runtime (simplified)

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

**However:**

- no absolute addresses are stored in the executable code
  → decided at “load time”
- not all the executable code is stored in the executable file
  (e.g., dynamic libraries)
  → lazy binding using relocation tables (e.g., GOT and PLT)
- data memory can be dynamically allocated
- data can become code (and conversely . . .)
- etc.

→ the executable file should contain all the information required . . .

∃ standards executable formats: ELF (Linux), PE (Windows), etc.

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

**Rks:** **stripped** (no symbol table) vs **verbose** (debug info) executables . . .
Example 1: Linux Elf

**ELF object file format**

<table>
<thead>
<tr>
<th>ELF header</th>
<th>Section header table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
<td></td>
</tr>
<tr>
<td>.text</td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td></td>
</tr>
<tr>
<td>.rodata</td>
<td></td>
</tr>
<tr>
<td>.bss</td>
<td></td>
</tr>
<tr>
<td>.sym</td>
<td></td>
</tr>
<tr>
<td>.rel.text</td>
<td></td>
</tr>
<tr>
<td>.rel.data</td>
<td></td>
</tr>
<tr>
<td>.rel.rodata</td>
<td></td>
</tr>
<tr>
<td>.line</td>
<td></td>
</tr>
<tr>
<td>.debug</td>
<td></td>
</tr>
<tr>
<td>.strtab</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: Windows PE
x86_64 assembly language in one slide

 Registers:
 ▶ stack pointer (RSP), frame pointer (RBP), program counter (RIP)
 ▶ general purpose: RAX, RBX, RCX, RDX, RSI, RDI
 ▶ flags

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

 Adressing modes (AT&T syntax):
 ▶ register: movl %rax, %rbx // rbx ← rax
 ▶ immediate: movl $1, %rax // rax ← 1
 ▶ direct memory: movl %rax, -0x10(%rbp) // Mem[rbp-16] ← rax
Stack layout for the x86 64-bits architecture (1)
Stack layout for the x86 64-bits architecture (2)

0x526 <assign>:
  0x526 push %rbp
  0x527 mov %rsp, %rbp
  0x52a mov $0x28, -0x4(%rbp)
  0x531 mov -0x4(%rbp), %eax
  0x534 pop %rbp
  0x535 retq

0x542 <main>:
  0x542 push %rbp
  0x543 mov %rsp, %rbp
  0x546 sub $0x10, %rsp
  0x54a callq 0x526 <assign>
  0x55f callq 0x536 <adder>
  0x554 mov %eax, -0x4(%rbp)
  0x557 mov -0x4(%rbp), %eax
  0x55a mov %eax, %esi

<table>
<thead>
<tr>
<th>Registers</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x0</td>
</tr>
<tr>
<td>%edi</td>
<td>1</td>
</tr>
<tr>
<td>%rsp</td>
<td>0xd20</td>
</tr>
<tr>
<td>%rbp</td>
<td>0xd20</td>
</tr>
<tr>
<td>%rip</td>
<td>0x531</td>
</tr>
</tbody>
</table>

Terminal:
$ ./prog
ABI (Application Binary Interface)

to “standardize” how processor resources should be used
⇒ 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.

<table>
<thead>
<tr>
<th></th>
<th>Cleans Stack</th>
<th>Arguments</th>
<th>Arg Ordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>cdecl</td>
<td>Caller</td>
<td>On the Stack</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>fastcall</td>
<td>Callee</td>
<td>ECX,EDX, then stack</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>stdcall</td>
<td>Callee</td>
<td>On the Stack</td>
<td>Left-to-Right</td>
</tr>
<tr>
<td>VC++ thiscall</td>
<td>Callee</td>
<td>EDX (this), then stack</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>GCC thiscall</td>
<td>Caller</td>
<td>On the Stack (this pointer first)</td>
<td>Right-to-left</td>
</tr>
</tbody>
</table>

**Figure:** some calling conventions

**System V AMD64 calling convention (Linux):**
Integer/Pointer Arguments 1-6 transmitted on RDI, RSI, RDX, RCX, R8, R9
Outline

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools ...
Understanding and analysing binary code?

Disassembling!

- **statically:** disassemble the *whole* file content *without executing it* . . .
- **dynamically:** disassemble the *current* instruction path *during execution/emulation* . . .
Static Disassembling (1)

Assume “reasonnable” (stripped) code only
   → no obfuscation, no packing, no auto-modification, ...

Enough pitfalls to make it undecidable ...

**main issue:** distinguishing code vs data ...

- interleavings between code and data segments
- dynamic jumps (`jmp <register>`)  
- possible variable-length instruction encoding, # addressing modes, ...
  e.g, > 1000 distinct x86 instructions
  1.5 year to fix the semantics of x86 shift instruction at CMU

→ much worse when considering **self-modifying code, packers**, etc.

**Example: x86 instruction format**
Static Disassembling (2)

Classical static disassembling techniques

- linear sweep: follows increasing addresses (ex: objdump)
  ↞ pb with interleaved code/data ?
- recursive disassembly: control-flow driven (ex: IDAPro)
  ↞ pb with dynamic jumps ?
- hybrid: combines both to better detect errors ...

Some existing tools

- Disassemblers/Decompilers:
  - IDA Pro [HexRays]
  - Ghidra [NSA, open-source]
- On Linux platforms (for ELF formats):
  - objdump (-S for code disassembling)
  - readelf
- and many others (Capstone, Miasm, Radare2, Triton, etc.)
- ... + a huge number of utility tools
  (hexadecimal operations, executable dissectors, etc.)
See some Emmanuel Fleury slides . . .
Indirect Jumps

BRANCH $R_i$

(branch address computed at runtime and stored inside register $R_i$)

⇒ A critical issue for static disassemblers/analysers ...

Occurs when compiling:

▶ some swicth statements
▶ high-order functions (with function as parameters and/or return values)
▶ pointers to functions
▶ dynamic method binding in OO-languages, virtual calls
▶ etc.
Example of Indirect Jump (borrowed from E. Fleury)

Source code example:

```c
enum {DIGIT, AT, BANG, MINUS}
f (char c) {
switch(c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9': return DIGIT ;
case '@': return AT ;
case '!': return BANG ;
case '-': return MINUS ;
}
}
```

Code produced with x86-64 gcc8.2

```assembly
f:
    push   rbp
    mov    rbp, rsp
    mov    eax, edi
    mov    BYTE PTR [rbp-4], al
    movsx  eax, BYTE PTR [rbp-4]
    sub     eax, 33 ; Ascii for '!' 
    cmp     eax, 31 ; 64 is Ascii for '@' 
    ja .L2 ; out of bounds ...
    mov    eax, eax
    mov    rax, QWORD PTR .L4[0+rax*8] ; offset in a jump table
    jmp    rax
```

3See https://godbolt.org/
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

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools...
Reminder

A classical buffer overflow situation . . .
▶ the content of the target buffer is attacker controlled
▶ the return address can be overwritten (no protections)
▶ the control-flow can be re-directed to a **shell code**

Remaining questions:
▶ where to put the shell-code ?
▶ which “input value” should be provided by the attacker ?
Writting the shell-code in the stack (1)

**Solution 1:** put the shell-code **below** the return address (i.e., in the caller’s stack frame)

attacker input = padding + \[\textcolor{red}{\text{shellAddr}}\] + **shell-code**
**Solution 2:** put the shell-code inside the **target buffer** (i.e., in the current stack frame)

- attacker input = **shell-code** + padding + **shellAddr**

- Diagram:
  - `shellAddr` (=`buffAdr`)
  - `target buffer`
  - `saved EBP`
  - `return address`
  - `shell−code`
  - `shellAddr`
When the stack segment is not executable?

Do not store **shellcode** in the stack . . . use **existing code instructions** instead!

- **return-to-libc**: redirect the control-flow towards library code
- **return oriented programming** (ROP)
  
  payload = sequence of return-terminated instructions (**gadgets**)

- gadget programming is “turing complete”
- ∃ tools for gadget extraction (ROPgadget, Ropper, ROPium, etc.)
- ∃ ROP variants:
  - COP (**call-oriented** programming), JOP (**jump-oriented** programming)

**Rks**: may also ∃ library calls allowing to **make the stack executable** . . .
Outline

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools
Objectives

When the code has been (partially !) disassembled . . .

. . . how to retrieve useful source-level information ?
(e.g.: variables, types, functions, control and data-flow relations, etc.)

Challenges

Still a gap between assembly and source-level code . . .

▶ basic source elements lost in translation:
  functions, variables, types, (conditionnal) expressions, . . .
▶ pervasive address computations (addresses = values)
▶ etc.

Rk: ≠ between code produced by a compiler and written by hand
  (structural patterns, calling conventions, . . .)

Again, ∃ static and dynamic approaches . . .
Function identification

Retrieve functions boundaries in a stripped binary code?

Why is it difficult?

▶ not always clean call/ret patterns:
optimizations, multiple entry points, inlining, etc.
▶ not always clean code segment layout:
extra bytes (∉ any function), non-contiguous functions, etc.

Possible solution...

▶ from pattern-matching on (manually generated) binary signatures
  ▶ simple ones (push [ebp]) or advanced heuristics as in [IDAPro]
  ▶ standart library function signature database (FLIRT)
▶ ...
▶ to supervised machine learning classification...

→ no “sound and complete” solutions...
Variable and type recovery

2 main issues
▶ retrieve the memory layout (stack frames, heap structure, etc.)
▶ infer size and (basic) type of each accessed memory location

Memory Layout
“addresses” of global/local variables, parameters, allocated chunks
▶ static basic access patterns \((epb+offset)\) [IDAPro]
▶ Value-Set-Analysis (VSA)

Types
▶ dynamic analysis:
  type chunks (library calls) + loop pattern analysis (arrays)
▶ static analysis: VSA + Abstract Structure Identification
▶ Proof-based decompilation relation inference
  type system + program witness [POPL 2016]
Static variable recovery

Retrieve the **address** (and size) of each program “variable”?

**Difficult because:**
- addresses and other values are not distinguishable
- address $\leftrightarrow$ variable is not one-to-one
- address arithmetic is pervasive
- both direct and indirect memory addressing

**Memory regions + abstract locations**

A memory model with 3 distinct regions:
- **Global**: global variables
- **Local**: local variables + parameters (1 per proc.)
- **Dynamic**: dynamically allocated chunks
- **Registers**

$\leftrightarrow$ associates a relative address to each variable (**a-loc**)
The so-called “naive” approach (IDAPro)

Heuristic
Adresses used for direct variable accesses are:
- absolute (for globals + dynamic)
- relative w.r.t frame/stack pointer (for globals)
→ can be statically retrieved with simple patterns . . .

Limitations
- variables indirectly accessed (e.g., \[eax\]) are not retrieved (e.g., structure fields)
- array = (large) contiguous block of data

⇒ Fast recovery technique, can be used as a bootstrap
But coarse-grained information, may hamper further analyses . . .
typedef struct {
    int i;
    char c;
} S;

int main() {
    S x, a[10];
    char *p1; int *p2;
    p1 = &(a[9].c);
    p2 = &(x.i);
    return 0;
}

<table>
<thead>
<tr>
<th>a</th>
<th>-60</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.i</td>
<td>-10</td>
</tr>
<tr>
<td>p2</td>
<td>-8</td>
</tr>
<tr>
<td>p1</td>
<td>-4</td>
</tr>
</tbody>
</table>

var_60 = byte ptr -60h
var_10 = byte ptr -10h
var_8  = dword ptr -8
var_4  = dword ptr -4

push ebp
mov ebp, esp
sub esp, 60h
lea eax, [ebp+var_60]
add eax, 4Ch
mov [ebp+var_4], eax
lea eax, [ebp+var_10]
mov [ebp+var_8], eax
mov eax, 0
leave
retn

main endp
Going beyond: Value Set Analysis (VSA)

Compute the contents of each a-loc at each program location . . .

. . . as an **over-approximation** of:

- the set of (integer) values of each data at each prog. loc.
- the addresses of “new” a-locs (indirectly accessed)

→ combines simultaneously numeric and pointer-analysis

**Rk:** should be also combined with CFG-recovery . . .

⇒ Can be expressed as a forward data-flow analysis . . .

A building block for many other static analysis . . .

- function “signature” (size and number of parameters)
- data-flow dependencies, taint analysis
- alias analysis
- type recovery, abstract structure identification
- etc.
Example: data-flow analysis

Does the value of $y$ depend from $x$?

```c
int x, *p, y;
x = 3;
p = &x;
...
y = *p + 4; // data-flow from x to y?
```

At assembly level:

1. needs to **retrieve** $x$ address
2. needs to **follow** memory transfers from $x$ address...

```assembly
mov [ebp-4], 3 /* x=3; */
lea eax, [ebp-4]
mov [ebp-8], eax /* p = &x; */
mov eax, [ebp-8]

... /* follow operations on eax ... */
mov eax, [eax] /* y = *p+4; ??? */
add eax, 4
mov [ebp-12], eax
```
CFG construction

Main issue
handling dynamic jumps (e.g., `jmp eax`) due to:

▶ switch statements ("jump table")
▶ function pointers, trampoline, object-oriented source code, . . .

Some existing solutions

▶ heuristic-based approach ("simple" switch statements) [IDA]
▶ abstract interpretation: interleaving between VSA and CFG expansion
  ▶ use of dedicated abstract domains
  ▶ use of under-approximations . . .

Rk: may create many program "entry points" ⇒ many CFGs . . .
Outline

Introduction

Low-level code representations

Disassembling

Application to BoF exploitation

Retrieving source-level information

Some Tools . . .
IDA Pro [HexRays]

- Commercial disassembler and debugger
- Supports 50+ processors (intel, ARM, .NET, PowerPC, MIPS, etc.)
- Recognizes library functions FLIRT (C/C++ only)
- Builds call graphs and CFGs
- Tags arguments/local variables
- Rename labels (variables names etc.)
- Provides scripting environment (IDC, Python) and debugging facilities
Script example

#include <idc.idc>
/* this IDA pro script enumerate all functions and prints info about them */
static main()
{
    auto addr, end, args, locals, frame, firstArg, name, ret;
    addr=0;
    for ( addr=NextFunction(addr); addr != BADADDR; addr=NextFunction(addr) )
    {
        name=Name(addr);
        end= GetFunctionAttr(addr,FUNCATTR_END);
        locals=GetFunctionAttr(addr,FUNCATTR_FRSIZE);
        frame=GetFunctionAttr(aiddr,FUNCATTR_FRAME);
        ret=GetMemberOffset(frame, " r");
        if (ret == -1) continue;
        firstArg=ret +4;
        args=GetStrucSize(frame) -firstArg;
        Message("function %s start at %x, end at %x\n",name, addr, end);
        Message("Local variables size is %d bytes\n",locals);
        Message("arguments size %d (%d arguments)\n",args, args/4);
    }
}
A dynamic code instrumentation framework

- run time instrumentation on the binary files
- provides APIs to define **insertion points** and **callbacks**
  (e.g., after specific inst., at each function entry point, etc.)
- Free for non-commercial use, works on Linux and windows
Example: instruction counting

```c
#include "pin.h"
UINT64 icount = 0;
void docount() { icount++; }

void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{ std::cerr << "Count " << icount << endl; }

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```