Software security, secure programming

Reverse-engineering from binary code

Master M2 Cybersecurity

Academic Year 2022 - 2023

Outline

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

“What You See Is Not What You Execute” [T. Reps]

- 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

Example 1: Java ByteCode (stack machine)[1]

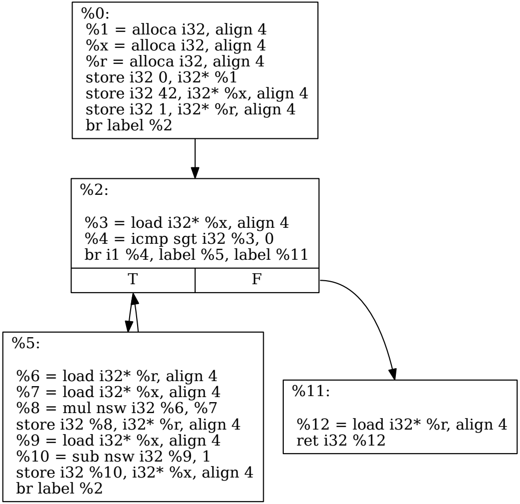
```
public static int main() {
int x, r;
x=42 ; r=1 ;
while (x>0) {
   r = r*x;
   x = x-1;
} ;
return r ;
}
```

```
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
```

[1] use `javap -c` to produce the bytecode

Example 2: LLVM IR (register based machine)

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



CFG for 'main' function

Example 3: assembly code (x86-64)[2]

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

```asm
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]
        pop     rbp
        ret
```

[2] see https://godbolt.org/

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)
  $\rightarrow$ lazy binding using relocation tables (e.g., GOT and PLT)
- data memory can be dynamically allocated
- data can become code (and conversely . . .)
- etc.

$\rightarrow$ the executable file should contain all the information required . . .

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

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

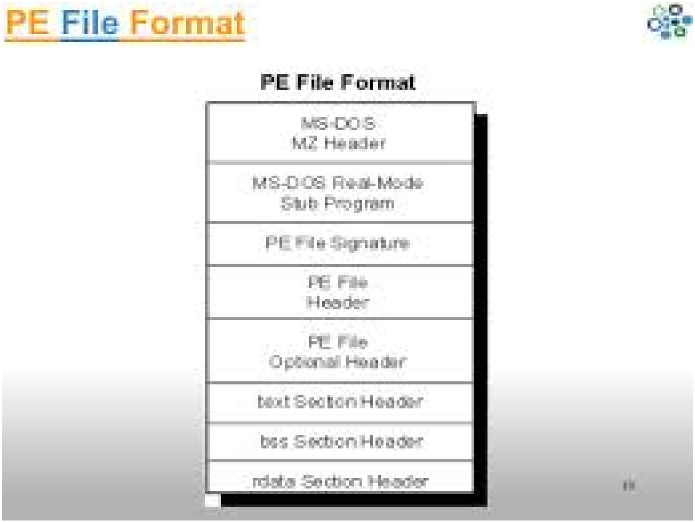
Rks: stripped (no symbol table) vs verbose (debug info) executables . . .

Example 1: Linux Elf

ELF object file format

ELF header
Program header table
.text
.data
.rodata
.bss
.sym
.rel.text
.rel.data
.rel.rodata
.line
.debug
.strtab
Section header 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

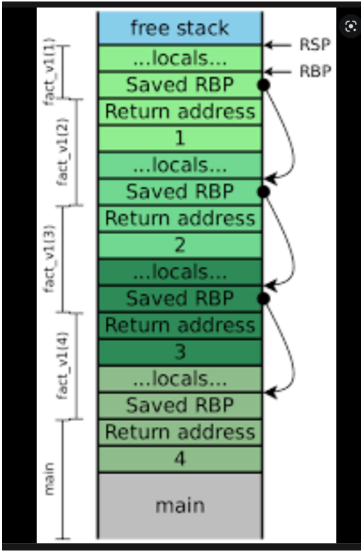
Instructions:

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

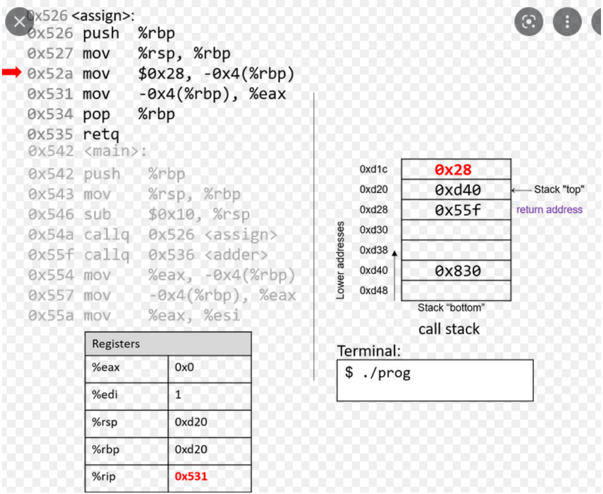
Adressing modes (AT&T syntax):

- register: movl %rax, %rbx // rbx $\leftarrow$ rax
- immediate: movl $1, %rax // rax $\leftarrow$ 1
- direct memory: movl %rax, -0x10(%rbp) // Mem[rbp-16] $\leftarrow$ rax

Stack layout for the x86 64-bits architecture (1)



Stack layout for the x86 64-bits architecture (2)



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

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

Outline

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 during execution/emulation . . .

Static Disassembling (1)

Assume “reasonnable” (stripped) code only

$\rightarrow$ no obfuscation, no packing, no auto-modification, . . .

Enough pitfalls to make it undecidable . . .

main issue: distinguishing code vs data . . .

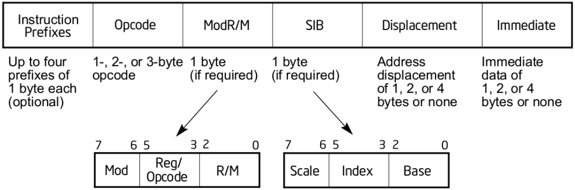
- interleavings between code and data segments
- dynamic jumps (jmp <register>)
- possible variable-length instruction encoding, # addressing modes, . . .

  e.g, $>$ 1000 distinct x86 instructions

  1.5 year to fix the semantics of x86 shift instruction at CMU

$\rightarrow$ 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 plateforms (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.)

Static disassembly (cont'd)

See some Emmanuel Fleury slides . . .

Indirect Jumps

BRANCH R_i

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

$\Rightarrow$ 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:

```
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`[3]

```
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
```

[3] See 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:

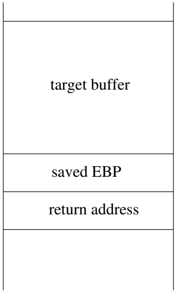
- only a (small) part of the executable is disassembled
- need some suitable **execution plateform**, e.g.:
  - emulation environment
  - binary level code instrumentation
  - (scriptable) debugger
  - etc.

Outline

Reminder

A classical buffer overflow sitation . . .

- 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 ?

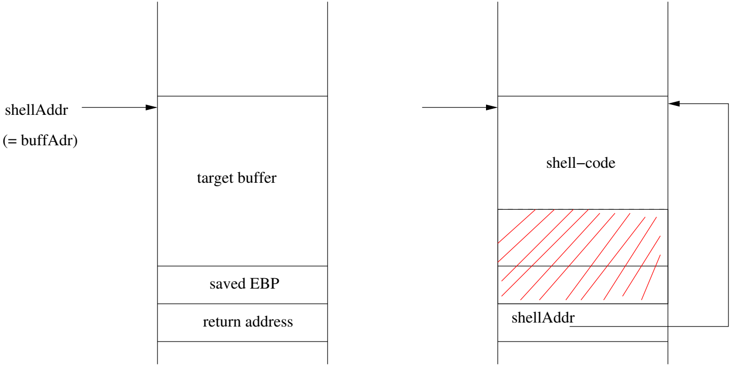
Writting the shell-code in the stack (1)

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

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

Writting the shell-code in the stack (2)

Solution 2: put the shell-code inside the **target buffer**
(i.e., in the current stack frame)

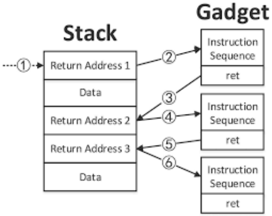


attacker input = **shell-code** + padding + shellAddr

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 gagdget 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

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: $\neq$ between code produced by a compiler and written by hand
(structural patterns, calling conventions, . . .)

Again, $\exists$ 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 ($\notin$ 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 paterns (`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 adresssing

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

$\hookrightarrow$ 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)

$\rightarrow$ can be statically retrieved with simple patterns . . .

Limitations

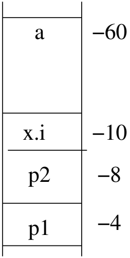
- variables indirectly accessed (e.g., `[eax]`) are not retrieved (e.g., structure fields)
- array = (large) contiguous block of data

$\Rightarrow$ Fast recovery technique, can be used as a bootstrap
But coarse-grained information, may hamper further analyses . . .

Example

```
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 ;
}
```




```
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)

$\rightarrow$ combines simultaneously numeric and pointer-analysis
Rk: should be also combined with CFG-recovery . . .

$\Rightarrow$ 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` ?

```
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 ...

```
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” $\Rightarrow$ many CFGs . . .

Outline

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 funtions 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);
  }
}
```

PIN [Intel]

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

```
#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;
}
```