Software security, secure programming

Reverse-engineering from binary code

Master M2 Cybersecurity

Academic Year 2023 - 2024

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)

Why and when bothering with binary code ? (1)

$\rightarrow$ 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)

$\rightarrow$ 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.)

Why and when bothering with binary code ? (2)

$\rightarrow$ 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]

```java
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
```
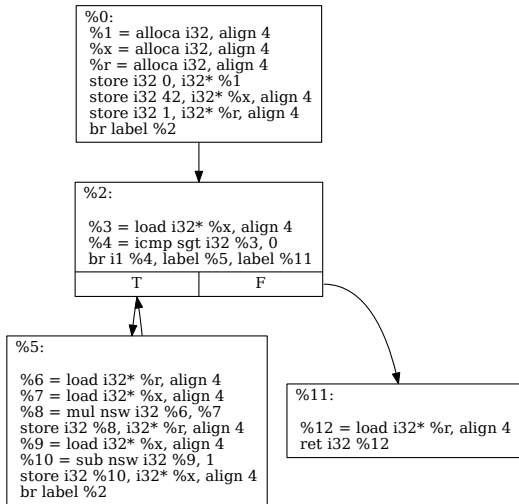
---

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

Example 2: LLVM IR (register based machine)

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

```
%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 | F |

```
%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
```

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

CFG for 'main' function

Example 3: assembly code (x86)[2]

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

```
main:
        pushl %ebp
        movl %esp,%ebp
        subl $8,%esp
        movl $42,-4(%ebp)
        movl $1,-8(%ebp)
.L2:
        cmpl $0,-4(%ebp)
        jle .L3
        movl -8(%ebp),%edx
        imull -4(%ebp),%edx
        movl %edx,-8(%ebp)
        decl -4(%ebp)
        jmp .L2
.L3:
        movl -8(%ebp),%eax
        jmp .L1
.L1:
        leave
        ret
```

---

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

Example 4: assembly code (x86-64)[3]

```
                           main:
                                   push    rbp
                                   mov     rbp, rsp
                                   mov     DWORD PTR [rbp-4], 42
                                   mov     DWORD PTR [rbp-8], 1
   int x, r;                       jmp     .L2
   x=42 ; r=1 ;          .L3:
      r = r*x;                     mov     eax, DWORD PTR [rbp-8]
      x = x-1;                     imul    eax, DWORD PTR [rbp-4]
   } ;                             mov     DWORD PTR [rbp-8], eax
   return r ;                      sub     DWORD PTR [rbp-4], 1
   }                     .L2:
                                   cmp     DWORD PTR [rbp-4], 0
                                   jg      .L3
                                   mov     eax, DWORD PTR [rbp-8]
                                   pop     rbp
                                   ret
```

---

[3]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)

▶ data memory can be dynamically allocated

▶ data can become code (and conversely . . .)

▶ etc.

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

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 abolute 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)
- ▶ 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/unitialized data, symbol tables, relocation
  tables, etc.

Rks: stripped (no symbol table) vs verbose (debug info) executables . . .
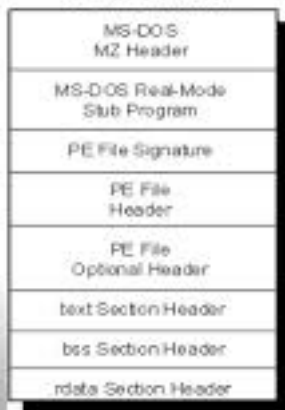
ELF object file format

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

PE File Format

PE File Format

MS-DOS MZ Header
MS-DOS Real-Mode Stub Program
PE File Signature
PE File Header
PE File Optional Header
text Section Header
bss Section Header
rdata Section Header

x86 (32) assembly language in one slide

Registers:

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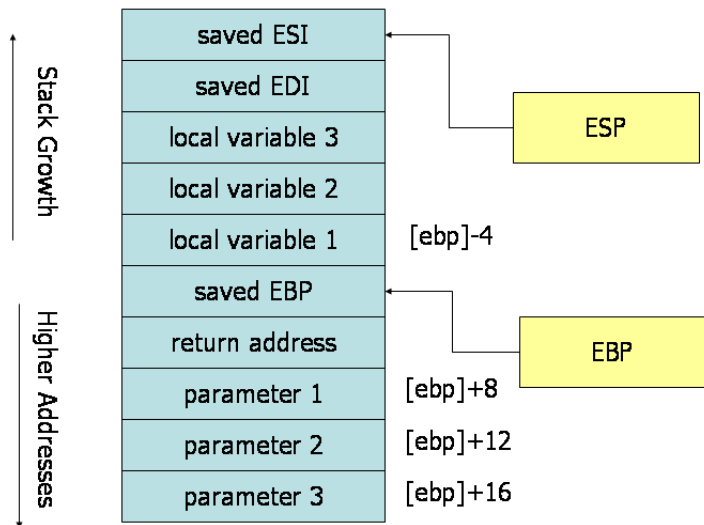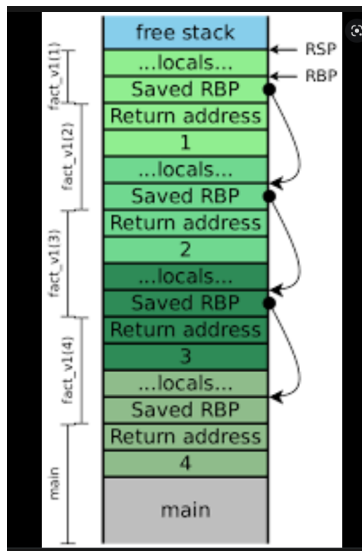- ▶ general purpose: EAX, EBX, ECX, EDX, ESI, EDI
- ▶ flags

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

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

```
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
```

Registers	
%eax	0x0
%edi	1
%rsp	0xd20
%rbp	0xd20
%rip	0x531

0xd1c	**0x28**
0xd20	0xd40
0xd28	0x55f
0xd30	
0xd38	
0xd40	0x830
0xd48	

Stack "top"
return address
Stack "bottom"

Lower addresses

call stack
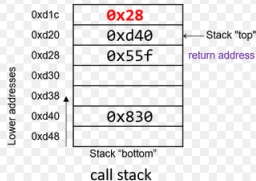
Terminal:
```
$ ./prog
```

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

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

Understanding and analysing binary code ?


```
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

Static Disassembling (1)

Assume "reasonnable" (stripped) code only
$\rightarrow$ no obfuscation, no packing, no auto-modification, . . .
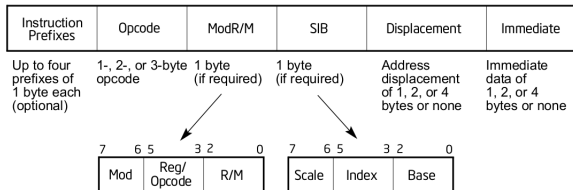
Enough pitfalls to make it undecidable . . .

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

Instruction Prefixes	Opcode	ModR/M	SIB	Displacement	Immediate
Up to four prefixes of 1 byte each (optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes or none	Immediate data of 1, 2, or 4 bytes or none

7	6 5	3 2	0
Mod	Reg/ Opcode	R/M	

7	6 5	3 2	0
Scale	Index	Base	

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

- ▶ Ghidra
  Developed by the NSA, ∃ a free version, # useful features
- ▶ IDA Pro
  a well-known commercial disassembler, # useful features
- ▶ On Linux plateforms (for ELF formats):
  - ▶ `objdump` (`-S` for code disassembling)
  - ▶ `readelf`
- ▶ and many others (Capstone, Radare2, Miasm, etc.)

Rk: may produce assembly-level IR instead of native assembly code
→ simpler language (a few instruction opcodes), explicit semantics (no side-effects), share analysis back-ends

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

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 ;
}
}
```

---

[4]See https://godbolt.org/

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`[4]

```
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
```

---

[4]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

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 ($\not\in$ 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

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)

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 ↔ variable is not one-to-one
- ▶ address arithmetic is pervasive
- ▶ both direct and indirect memory adressing

Static variable recovery

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

Difficult because:
- ▶ addresses and other values are not distinguishable
- ▶ address ↔ 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

↪ 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)
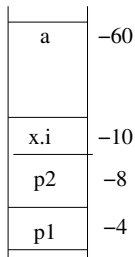- ▶ array = (large) contiguous block of data

$\Rightarrow$ 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 ;
}
```

a	−60
x.i	−10
p2	−8
p1	−4

```
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 . . .

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 . . .

$\boxed{\Rightarrow \text{Can be expressed as a forward data-flow analysis} \ldots}$

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

An (ultra) lightweight dynamic technique

Starting from a binary code . . .

- ▶ without source, debug information, symbol table
- ▶ but those architecture and calling convention is known
- ▶ and which can be instrumented & executed

. . . retrieve function-level information

- ▶ function arity and signatures
- ▶ quantified coarse grain data-flow information between functions
- → within a **single code execution**

General approach

A 3-steps process

1. a lightweight dedicated binary code instrumentation to collect runtime information
2. the one trace execution step to generate a log file
3. an offline log analysis to produce the results . . .

Relying on aggressive heuristics to approximate the notion of parameter, type and data-flow . . .

Main heuristics

parameter definition

a memory location read before written is a input parameter
(holds also accross function boundaries)

Main heuristics

parameter definition

a memory location read before written is a input parameter
(holds also accross function boundaries)

type definition

- ▶ *ADDR* types can be deduced from load/store operations
- ▶ once an *ADDR*, always an *ADDR*
- ▶ non *ADDR* values are of type *NUM*

Main heuristics

parameter definition
a memory location read before written is a input parameter
(holds also accross function boundaries)

type definition
- ▶ *ADDR* types can be deduced from load/store operations
- ▶ once an *ADDR*, always an *ADDR*
- ▶ non *ADDR* values are of type *NUM*

data-flow definition
- ▶ consider only *ADDR* flows
- ▶ *ADDR* collisions are not fortuitous:
  ADRR value *a* produced by *foo* and consumed by *bar* $\Rightarrow$ data-flow fron *foo* to *bar* ...

Implementation

SCAT, open source: `https://github.com/Frky/scat`

- ▶ dynamic code instrumentation using PIN
  $\rightarrow$ function detection based on call/ret instructions

- ▶ minimize the size of the instrumentation code
  $\rightarrow$ extra implementation level heuristics
  (e.g., a value betteen two *ADDR* is an *ADDR*)

- ▶ user given *MIN_CALL* threshold

- ▶ embeds an oracle[5] for function signatures

Experiments:
- ▶ coreutils (> 100 pgms)
- ▶ 10 common Linux pgms: `git`, `grep`, `mupdf`, `objdump`, `openssl`, etc.

---

[5]based on `clang`

Experimental results: arity

arity

- EVALUATION - PARAMETERS -

PROGRAM	accuracy	fn	fp	total	overhead
8cc	98 %	4	2	283	5,43
bash	95 %	8	7	283	1,92
coreutils	93 %	154	15	2515	3,67
git	96 %	15	6	492	2,37
grep	91 %	6	2	86	3,46
mupdf	94 %	13	7	348	4,55
objdump	89 %	7	7	132	4,10
openssl	96 %	4	3	194	3,04
opusenc	94 %	2	0	36	10,97
...	...	...	...	...	
TOTAL	**93 %**	248	61	4546	**3,76**
	(average)	(sum)	(sum)	(sum)	(average)

Experimental results: types

- EVALUATION - PARAMETERS -

PROGRAM	accuracy	fn	fp	total	overhead
8cc	98 %	4	3	307	6,37
bash	96 %	14	2	374	1,75
coreutils	92 %	208	55	3299	2,51
git	94 %	26	4	530	2,12
grep	95 %	4	3	129	2,99
mupdf	96 %	17	11	746	7,12
objdump	94 %	3	12	231	3,59
openssl	95 %	9	5	308	2,99
opusenc	98 %	0	1	53	7,23
...	...	...	...	...	
TOTAL	**96 %**	296	101	6294	**3,53**

47

Outline

IDA Pro [HexRays]

A swiss-knife for reverse engineering . . .

▶ 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 swiss-knife for binary-level dynamic analysis . . .

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;
}
```
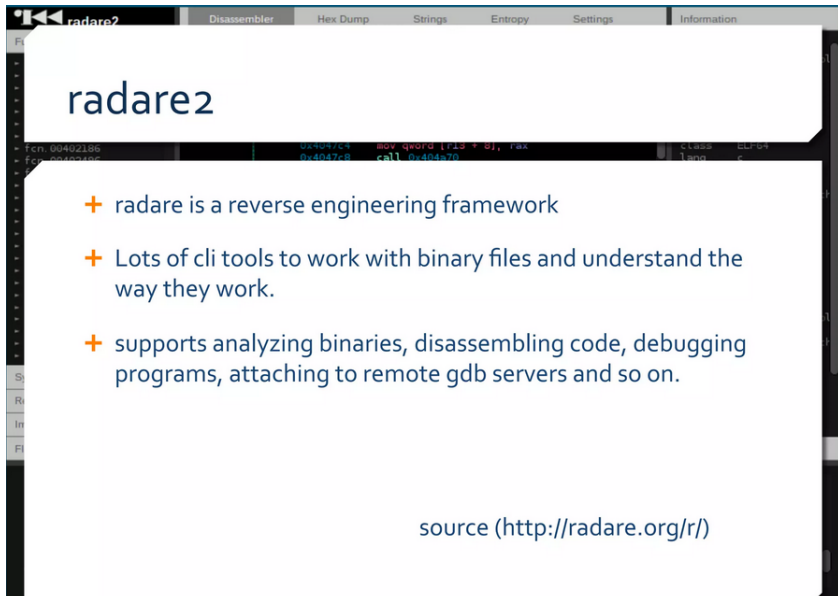
Radare2 [radare.org]

Use Cases

+ Disassemble (and assemble for) many different architectures

+ Debug with local native and remote debuggers (gdb, rap, webui, r2pipe, winedbg, windbg)

+ Perform forensics on file systems and data carving

+ Be scripted in Python, Javascript, Go and more

+ Support collaborative analysis using the embedded webserver

+ Visualize data structures of several file types

+ Patch programs to uncover new features or fix vulnerabilities

+ Use powerful analysis capabilities to speed up reversing

+ Aid in software exploitation

(source : http://radare.org/r/)

Radare2 [radare.org]

Support

+ radare2 supports:

+ Architectures 6502, 8051, arm, arc, avr, bf, tms320 (c54x, c55x, c55+), gameboy csr, dcpu16, dalvik, i8080, mips, m68k, mips, msil, snes, nios II, sh, sparc, rar, powerpc, i386, x86-64, H8/300, malbolge, T8200, LH5801

+ File formats bios, dex, elf, elf64, filesystem, java, fatmacho, macho, macho-64, MZ, PE, PE+, TE, COFF, plan9, bios, dyldcache, Gameboy and Nintendo DS ROMs

+ Operating systems Android, GNU/Linux, [Net|Free|Open]BSD, iOS, OSX, QNX, w32, w64, Solaris, Haiku, FirefoxOS

+ Bindings Vala/Genie, Python (2, 3), NodeJS, LUA, Go, Perl, Guile, php5, newlisp, Ruby, Java, OCAML and more