UFR IM²AG

UNIVERSITÉ Grenoble Alpes

Grenoble INP Ensimag

Software security, secure programming (and computer forensics)

Lecture 4: an overview of Software Security Analysis Techniques

Master M2 on Cybersecurity

Academic Year 2016 - 2017

Software Security

The ability of a SW to *function correctly* under *malicious attacks*

"function correctly" ?

- no crash (!), no disclosure/erasure of confidential data
- no bypass of security policy rules
- no deviation from intended behavior (arbitrary code execution)

$\rightarrow$ what the SW should **not** do ...

"malicious attacks" ?

Well-crafted attack vectors, based on knowledge about:

- execution platform: libraries, OS/HW protections
- target software: code, patches
- up-to-date vulnerabilities and exploit techniques

$\rightarrow$ much beyond unexpected input/execution conditions

secure software $\neq$ robust/safe/fault-tolerant software

Root causes of insecure softwares

"A software flaw that may become a security threat . . . "

$\neq$ kinds of bugs w.r.t security:

- harmless: only leads to incorrect results or "simple" crash
- **exploitable**: can lead to unsecure behaviors . . .

Examples of exploitable vulnerabilities

(combinations of:)

- invalid memory accesses: buffer overflow, dangling pointers
- arithmetic overflows
- race conditions
- etc.

Rk: influence of programming language, compilation tool,
execution environment (plateform, OS, users . . .)

Vulnerability detection and analysis

A major security concern . . .

- 5000 vulns in 2011, 5200 in 2012, 6700 in 2013 ... [Symantec]
- applications and OS editors, security agencies, defense departments, IT companies, . . .

. . . and a business !

Some 0-day selling prices [Forbes, 2012]:
Adobe Reader : $30,000 - Chrome, IE: $200,000 - ios : $250,000

Two distinct problems

1. detection: identify (security related) bugs
2. analysis: evaluate their dangerousness
   Are they exploitable? How difficult is it? Which consequences?

The current "industrial" practice

A 2-phase approach

1. (pseudo-random) fuzzing, fuzzing, and fuzzing . . .
   $\hookrightarrow$ to produce a huge number of **program crashes**
2. in-depth ***manual*** crash analysis
   $\hookrightarrow$ to identify **exploitable** bugs and obtain PoC exploits (ignoring protections)

Drawbacks

- A time consuming activity
  (very small ratio "exploitable flaws/simple bugs" !)
  100,000 open bugs for Linux Ubuntu ; 8000 for Firefox
- Would require a better **tool assistance** . . .
  (e.g., "smart" disassembler, trace analysis, debuggers ?)

example: crash of `/bin/make` on Linux . . .

The “academic” research trends

Re-use and adapt **validation oriented** code analysis techniques

- static analysis, bounded model-checking
- test generation:
  symbolic/concolic execution, genetic algos, etc.
- dynamic (trace based) analysis

security analysis $\neq$ safety analysis !

- should be carried on the executable code
- exploit analysis $\Rightarrow$ **beyond** source-level semantics
  (understand what can happen **after** an undefined behavior)

Main issue: **scalability !** . . .

DARPA CGC: software security tool competition (1st prize: $2,000.000)

Outline

Software Security

Outline of the next parts of the course

Disassembling

Oral presentations

Some security-oriented code analysis techniques

- **Disassembling**
  from binary code to assembly-level code representation
- **Fuzzing**
  how to make a program crash ?
- **Static Analysis**
  analyse an approximation of the code behaviour without executing it
- **Dynamic Analysis**
  collect (more) useful information at runtime
- **(Dynamic) Symbolic Execution (DSE)**
  explore a (comprehensive) subset of the execution sequences

And, in addition, an overview of:

- code (de)-obfuscation techniques
- code hardening

Course organization

- lectures
- paper exercises
- lab sessions (on tools)
  static analysis, DSE, fuzzing, code instrumentation, . . .
- oral presentations (see later)

Outline

Understanding binary code ? (1/2)

```
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 !

Understanding and analysing binary code ? (2/2)

Recovering assembly-level code

- a non trivial task (static disassembling of x86 code undecidable)
- may produce assembly-level IR ($\neq$ native assembly code)
  $\rightarrow$ simpler language (a few instruction opcodes), explicit semantics (no side-effects), share analysis back-ends

Handling assembly-level code

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

- recovering basic program elements:
  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, ...)

Static Disassembling

Assume “reasonnable” (stripped) code only

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

Enough pitfalls to make it undecidable ...

- interleavings between code and data segments
- dynamic jumps (`jmp eax`)
- variable-length instruction encoding, # addressing modes, ...

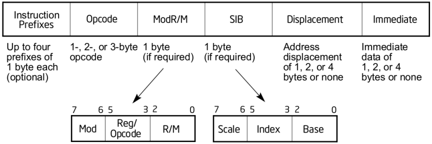
  e.g, $>$ 1000 distinct x86 instructions

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

Classical static disassembling techniques

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

Instruction encoding: the x86 example



r8(/r)			AL	CL	DL	BL	AH	CH	DH	BH
r16(/r)			AX	CX	DX	BX	SP	BP	SI	DI
r32(/r)			EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
mm(/r)			MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
xmm(/r)			XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
sreg			ES	CS	SS	DS	FS	GS	res.	res.
eee			CR0	invd	CR2	CR3	CR4	invd	invd	invd
eee			DR0	DR1	DR2	DR3	DR4[1]	DR5[1]	DR6	DR7
(In decimal) /digit (Opcode)			0	1	2	3	4	5	6	7
(In binary) REG =			000	001	010	011	100	101	110	111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hex)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[sib]		100	04	0C	14	1C	24	2C	34	3C
disp32		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[sib]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[sib]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
AL/AX/EAX/ST0/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
CL/CX/ECX/ST1/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
DL/DX/EDX/ST2/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
BL/BX/EBX/ST3/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB

Function identification

Retrieve functions boundaries in a stripped binary code ?

Why is it difficult ?

- not always clean `call/ret` patterns:
  optimizations, multiple entry points, inlinning, etc.
- not always clean code segment layout:
  extra bytes ($\notin$ any function), non-contiguous functions, etc.

Possible solution . . .

- pattern-matching on (manually generated) binary signatures
  - simple ones (`push [ebp]`) + proprietary heuristics [`IDA`, `Bap`]
  - standart library function signature database (FLIRT)
- supervised machine learning classification

$\boxed{\rightarrow \textbf{no ``sound and complete'' solutions . . .}}$

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]
- lightweight static analysis (e.g., intraprocedural data-flow)
- 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]

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]
- static analysis: interleaving between VSA and CFG expansion
  - stridded-intervals vs k-sets, refinment-based approach
  - use of under-approximations ...

Rk: may create many program "entry points" $\Rightarrow$ many CFGs ...

To continue:

- Control-Flow Graphs (CFG) and intermediate representation (IR)
- Some basics on x86
- More on disassembling and reverse-engineering
  - disassembling techniques (and their limitations)
  - tools
  - examples

Outline

Software Security

Outline of the next parts of the course

Disassembling

Oral presentations

Suggested topics (a non limitative list !)

- Programming languages and/or execution plateforms
  $\hookrightarrow$ focus on specific features, explain the strenght/weaknesses, and the associated protections . . .
  - Java / JVM / Android / . . .
  - Rust
  - JavaScript / PhP / web / . . .
- Protections
  - Control-Flow Integrity (CFI)
  - Windows 10 protections
- Malwares
  principles, detection and identification techniques
- Code (de)-obfuscation techniques
- Vulnerability exploitation techniques
  Return-Oriented-Programming (ROP), defeating ASLR, etc.
- Vulnerability on cryptographic functions implementations

Organisation

One oral presentation per "binôme" (team of 2 students)

schedule:

- one 1.30 hour course slot **[before end of november]**
  choose and refine your subject, select resources (docs, tools) on the web
  $\rightarrow$ give back 1-2 slides with your subject outline + selected resources
- one 1.30 hour course slot **[before end of december]**
  discussion about your work
- 3 hours course dedicated to oral presentations **[before mid-january]**
  - 15 mn. presentation per binômes (with slides)
  - **a written report** (3-5 pages)